Arbeidsnotater fra Høgskolen i Buskerud nr. 72

ARBEIDSNOTAT

ARBEIDSNOTAT

Objektorientert analyse og design med UML

Knut W. Hansson

HØGSKOLEN i Buskerud

Arbeidsnotater fra Høgskolen i Buskerud

Nr. 72

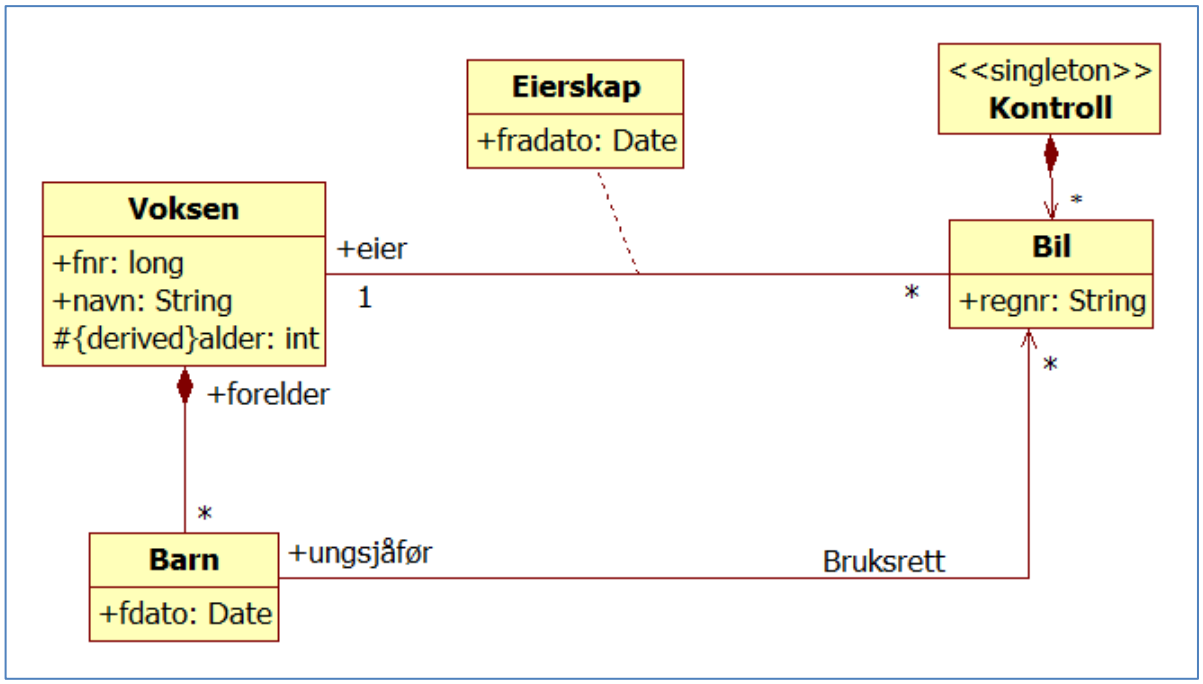
Objektorientert analyse og design med UML

Av

Knut W. Hansson

Hønefoss 2013

ISSN 1893-2398 (online)



Objektorientert analyse og design

med UML

Knut W. Hansson
Førstelektor IT
Høgskolen i Buskerud

Hønefoss, 16. mai 2013

Sammendrag
Ved høyskolen i Buskerud, bachelorstudiene i IT, undervises kurset "Objektorientert analyse og design" som gir 7,5 studiepoeng. *Denne boken inneholder alle forelesningene mine i det nevnte kurset, samt noe tilleggsstoff for spesielt interesserte.*

Det nevnte kurset bygger på et underliggende kurs i systemarbeid, et databasekurs med datamodellering og et kurs i objektorientert programmering med Visual Basic. Et kurs med objektorientert programmering med Java går parallelt. I de underliggende kursene skal studentene ha blitt kjent med prinsippene for objektorientert programmering med klasser, instansiering, arv, informasjonsskjuling osv. Videre skal de være kjent med prinsippene for systemarbeid generelt og med databasemodellering.

Kurset "Objektorientert analyse og design" bygger altså på dette og skal bringe inn et objektorientert modelleringsspråk (UML) for analyse/design av objektorienterte systemer. Det legges spesiell vekt på modelleringsteknikkene i "Use Case View" og "Design View". Videre skal studentene lære å strukturere en relasjonsdatabase for persistens av et objektorientert system ("mapping").

Det er laget en egen elektronisk lærerressurs med løsningsforslag til mange av oppgavene.

Synopsis in English
At Buskerud College, the course "Object Oriented Analysis and Design" is part of the bachelor IT education and gives ECT 7.5 credits. *This book contains all my lectures in the course, and some additional subjects for interested students.*

The abovementioned course builds on an underlying course in system development, a database course with data modeling and a course in object oriented programming with Visual Basic. A course with object oriented programming in Java is held in parallel. In the underlying courses, the students should have been acquainted with the principles of object oriented programming with classes, instantiation, inheritance, information hiding etc. They should also be acquainted with the principles for system development in general and with data base modeling.

This course builds on this knowledge and aims to bring in an object oriented modeling language (UML) for analysis/design of object oriented systems. Special consideration is placed on "Use Case View" and "Design View". In addition, the students should learn to structure a relational database for persistence of an object oriented system ("mapping").

An electronic teacher's resource has been made with suggested solution for many of the assignments.

Innhold

Kapitel 1 – Introduksjon og bakgrunn

Introduksjon til boken

Denne boken inneholder alle mine forelesninger i kurset "Objektorientert analyse og design" ved Høgskolen i Buskerud. Det skal ikke være nødvendig med ytterligere litteratur for å følge dette kurset. Boken kan også leses på egen hånd.

Uansett om du følger det nevnte kurset eller leser boken på egen hånd, bør du gjøre *alle* oppgavene (det er én til hvert kapittel). Det er ved å *gjøre* det du har lest om at du lærer best. Det er også et større case i slutten av boken, som du bør legge mye arbeid i. Caset omhandler fagstoff fra kapitlene frem til og med kapittel 7. I høyskolekurset gjennomfører vi derfor caset omtrent midt i semesteret.

Forkunnskaper

Boken forutsetter at du er godt kjent med objektorienterte prinsipper som klasser, objekter, arv, instansiering og informasjonsskjuling. Mestrer du ikke disse prinsippene får du erfaringsmessig problemer med forståelsen.

Videre er det bruk for gode kunnskaper i datamodellering for relasjonsdatabaser, men ikke SQL. Boken bygger også på generelle kunnskaper om systemarbeid.

Nytteverdi

Det er en generell erfaring at det ikke er lønnsomt – til tider ikke en gang mulig – å programmere store objektorienterte systemer uten først å ha analysert problemet og designet en løsning. All systemutvikling vil derfor modellere systemet som skal lages før det forsøkes programmert. Et modelleringsspråk er derfor et viktig "verktøy i systemutviklerens verktøykasse". Analogt er det nok vanskelig å lage en bil uten tegninger. Programmering av store systemer er minst like vanskelig som å lage en ny type bil.

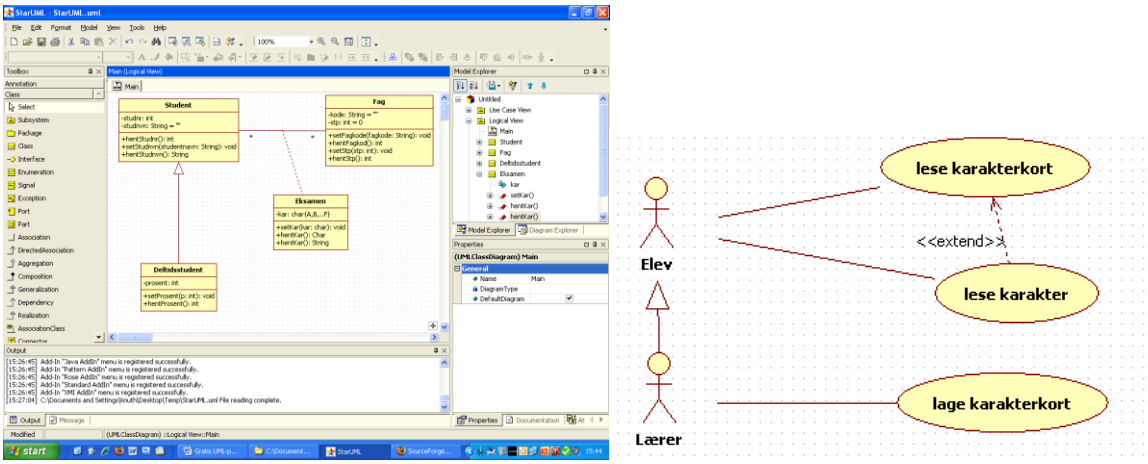
I denne boken skal jeg se på et modelleringsspråk utviklet spesielt for objektorienterte programmer.

I tillegg har kunnskaper om og erfaring med slik modellering betydelig overføringsverdi på objektorientert programmering. Under modelleringsarbeidet får man tenke på og dokumentere systemet uten å bli hektet fast i syntaksfeil i programmet og det er en klar fordel. Den forståelsen som det gir om hvordan klasser og objekter henger sammen, hvordan informasjon skjules osv., gjør det lettere å forstå koden som skal realisere modellen.

Verktøy

Det finnes mange verktøy – også gratis. Jeg har forsøkt en god del av dem og har landet på "StarUML". Du forutsettes å benytte dette verktøyet og evt. løsningsforslag vil bli publisert i StarUML-formatet (.uml). Det koster litt i starten å lære seg et verktøy, men StarUML er svært tro til teknikken i UML og derfor støtter det læringen på en fin måte.

StarUML er fri programvare under GPL lisens. Det hentes fra Sourceforge på http://staruml.sourceforge.net/en/download.php. Programmet er skrevet i Delphi og kompilert bare for Windows. Jeg har brukt det mye og er meget fornøyd, selv om det har vært lite vedlikeholdt i det siste. Det er komplett, med god UML-syntaks og med ryddig og greit grensesnitt. Be om ”Rational Approach” når du skaper ny modell, og sett den gjerne til default.



Eksempler på grafer fra StarUML

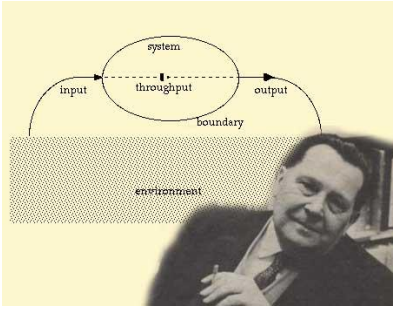
Diverse systemerings- og systembegreper

UML (Unified Modeling Language) har hentet teori og begreper fra en lang rekke kilder – det er bl.a. derfor de kaller det "Unified". Her samler jeg en rekke av dem i en assortert samling. Kanskje virker de urelaterte men de danner "bakteppe" og delvis begrunnelse for måten UML gjør ting på.

Systemering

Begrepet systemering ble skapt av professor Langefors (svensk). Han argumenterte for den "revolusjonerende" ideen at man måtte analysere og utforme systemer før man begynte å programmere dem. Ordet er laget av "system" og "programmering" og beskriver arbeidet med å planlegge systemet før det realiseres. Han laget også en diagramteknikk for å beskrive informasjonssystemer (informasjonsgraf). Langefors teorier fikk stort gjennomslag.

Universitetet i Oslo har tatt i bruk ordet "systemarbeid" som innebærer omtrent det samme, muligens med tillegg av prosjektstyring o.l. Dette begrepet brukes også ved vår høyskole.



Generell systemteori

Omkring 1950 utarbeidet von Bertalanffy[1] en "generell systemteori". På den tiden hadde stadig flere vitenskaper oppdaget at det ikke var tilstrekkelig å studere ett og ett object, ett og ett fenomen, men at de også måtte studeres *i sammenheng*. Helheten som objektet/fenomenet inngikk i ble like viktig som hver del. Sammenhenger som idag er opplagte, er at sykdom i leveren kan spre seg til nyrene, eller at et deprimert menneske påvirker humøret til andre rundt seg. Det er blitt meget vanlig i f.eks. kreftmedisin å behandle den sykes familie også (med andre midler naturligvis). Vitenskaper som hadde laget seg begreper om systemer var f.eks. psykologi, sosiologi og biologi. Siden begrepene var laget innenfor det enkelte fagmiljøet og ikke var like, var det vanskelig å kommunisere om dette på tvers av faggrensene.

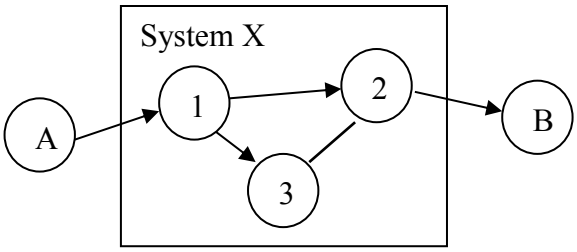
Von Bertalanffy var også opptatt av at systemene måtte ses i sammenheng med omgivelsene, da systemene som oftest påvirker og påvirkes av disse.

Von Bertalanffy mente at det var viktig å lage felles begreper for systemer. I tillegg skapte han nye. Han kalte samlingen av begreper – eller tankesettet om man vil – for generell

[1] Bildet til venstre er hentet fra http://psicoletra.blogspot.no/2012/06/sistemas-por-doquier-ludwig-von.html

systemteori (GST). Dette viste seg svært nyttig, og ble tatt i bruk både tilbake i de opprinnelige fagene og i alle(?) nye fag. Informasjonsteorien er én, beslutningsteori en annen osv.

Noen av begrepene fra GST[2] er gjengitt i denne figuren:



Systemer består av *deler* (1, 2 og 3 i figuren) med *relasjoner* (sammenhenger) mellom seg. Relasjonene kan være *retningsbestemt* (mellom 1 og 3 i figuren). Det innebærer i så fall at den ene (1) påvirker den andre (3) på en eller annen måte. Systemet har en *grense* (rektangelet i figuren) og utenfor systemet finner man dets *omgivelser* bestående av andre systemer som vårt system har relasjon til. Hver del kan være *elementær/atomær* (udelelig) eller et *subsystem* (da er system X delens *metasystem*).

Siden system X har relasjon til omgivelser, er det et *åpent* system. Systemer kan også være – eller betraktes som – *lukket*. Et system som utgjør en del av et annet, *må* være åpent.

Den som skal foreta en systemanalyse (*analysere* er å dele opp og finne sammenhenger), velger man selv systemgrense og velger deler som passer for formålet.

Artefakter

Artefakter er menneskeskapte ting. Ting lages av en grunn, dvs de har en hensikt.

Hensikten med et lagd system kan ligge i
- prosessen: Lage fordi det er lærerikt eller moro (elever/studenter)
- resultatet: Lage fordi det skal brukes til noe (privat eller av en organisasjon)
- begge deler: Prosessen er uansett hensikt alltid lærerik for dem som deltar – de lærer om interesseområdet (”universe of discourse”), om behov og muligheter, om dem som skal ha systemet, om andre deltakere i prosessen, om selve utviklingsprosessen og annet.

Artefakter vurderes etter i hvilken grad hensikten oppnås. Dette kalles "det teleologiske prinsipp".

Informasjonssystemer

- ✓ Informasjonssystemer er systemer som behandler informasjon.
- ✓ Informasjonssystemer er åpne. De omgivelsene som er mennesker, kalles interessenter og det er (noen av) dem som skal ha informasjonen.
- ✓ Informasjonssystemer er artefakter, og hvis hensikten ligger i resultatet, så er den å fremskaffe (nyttig) informasjon.
- ✓ Det er interessentene som må angi hvilken informasjon som er nyttig for dem.
- ✓ Det blir da viktig å finne ut hvilken hensikt interessentene har med systemet.

[2] GST inkluderer også andre systembegreper som selvregulering gjennom tilbakekobling (feedback), kommunikasjon og informasjon og annet. Her gjengir jeg bare det mest sentrale.

Langefors var av dem som benyttet GST. Han formulerte et "Fundamentalprinsipp for systemarbeid". Fundamentalprinsippet angir en metode for å studere store, "uoverblikkbare" systemer ved å dele dem opp i deler og studere hver del for seg. Deretter tar man en kontroll ved å sette delene sammen igjen og sammenlikne med den opprinnelige helheten. Hvis en del ikke er "overblikkbar" så deler man den opp i den igjen i nye deler (da blir den et subsystem) osv. inntil alle delene er overblikkbare. Jeg går ikke nærmere på dette prinsippet her men gjengir bare en liste over aktivitetene nedenfor.

Ved arbeid med uoverblikkbare systemer skal man følge disse momenter:

A Angivelse av systemets ytre egenskaper
Man angir systemets ytre egenskaper, dvs. de egenskaper ved systemet som systemets interessenter har angitt at det skal ha.

B Angivelse av systemets deler
Man deler systemet opp i de delene man antar det består av.

C Angivelse av relasjonene mellom delene
Man angir hvilke relasjoner det er mellom de delene man har delt systemet opp i.

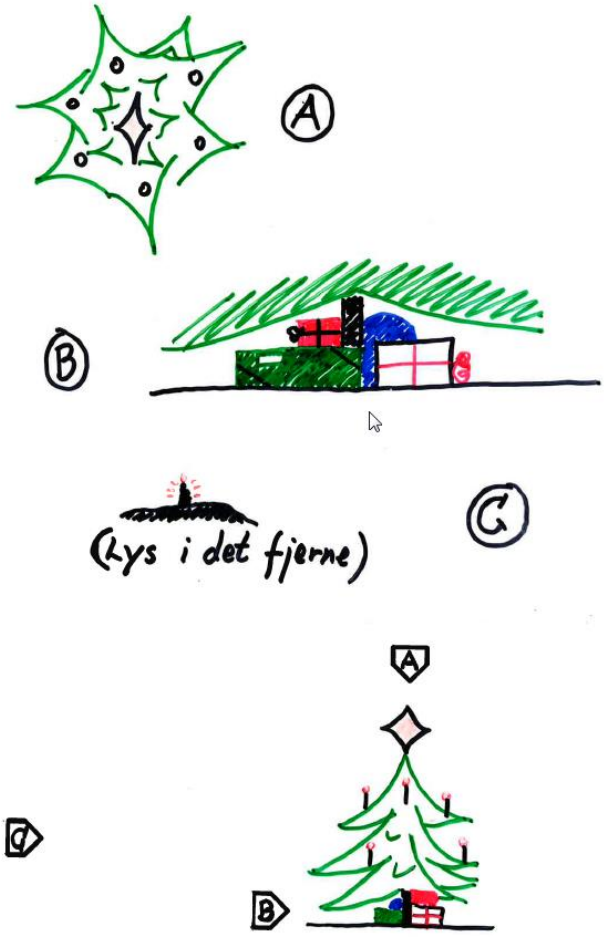
D Angivelse av hvert delsystems ytre egenskaper
Hver enkelt del betraktes som et nytt system. Man angir ytre egenskaper for hvert enkelt av disse delsystemene.

E Bestemmelse av delsystemstrukturens samlede ytre egenskaper og kontroll med at disse ytre egenskapene er de samme som for det opprinnelige systemet
Man foretar kontroll med at den innførte delsystemstrukturen (momentene B og C) og dennes totale ytre egenskaper (moment D) faller sammen med det opprinnelige systemets ytre egenskaper (moment A). Hvis vi ikke får et tilfredsstillende resultat, må momentene A–E gjennomgås på nytt, inntil man når fram til sammenfallende ytre egenskaper.

Langefors' "Fundamentalprinsipp for systemarbeid"

Perspektiver i systemering

Ordet perspektiv er dannet av *per* = gjennom og *spectere* =se. Det kan sammenliknes med å se gjennom noe. Det man ser vil bli "farget" av det man ser gjennom. I overført betydning betyr det "måte noe ser ut fra et bestemt sted eller synsvinkel". Det man ser avhenger av ståsted, men også av hva man ser etter. I nedenstående figur vises et juletre, slik det ser ut fra forskjellige kanter.



A er slik det tar seg ut for far som skal sette stjernen på toppen. *B* er slik Lillebror ser det og *C* er slik Storebror ser det på lang avstand når han kommer hjem fra Amerika. Alle er sanne bilder av det samme treet, det er bare sett fra forskjellige kanter og avstand og med forskjellig fokus. Legg f.eks. merke til hvor prominent pakkene fremkommer når Lillebror ser på treet. En botaniker – som ikke er tatt med her – vil antakelig legge spesielt merke til hvilken tresort det er og hvor sunt treet er. For å få et komplett bilde, må man se det fra alle *relevante* sider og med alle forskjellige fokus.

Selv deler jeg gjerne perspektivene i systemering i de intro- og de ekstrospektive, slik det fremgår nedenfor.

1. PERSPEKTIV: INTROSPEKTIVT

(Av *intro* inn - smlgn. *intra* innenfor - og *spektere* se)

✓ **Studere *oss selv* ... med sikte på å forbedre våre *arbeidsmåter/prosesser* altså hvordan man fremskaffer den informasjonen man trenger.**

Utg.pkt.:	**Prosesser** **Funksjoner** **Virksomheter**

Begynner med en modell av våre handlinger (de som er interessante).

DFD	**Terminatorene (datamålene) utenfor systemet krever visse data, hvilke data er det og hvordan lager man dem (dvs hvilke data må systemet få inn og hvorfra, hvilke dataprosesser kreves for å omforme dataene og hvilke datalagre bruker man mellom prosessene)?**
Rutinegraf	**Det oppstår hendelser som man må reagere på - hvilke er det og hva gjør man med dem?**

2. PERSPEKTIV: EKSTROSPEKTIVT

(Av *ekstro* ut - smlgn. *ekstra* utenfor - og *spektere* se)

✓ **Studere *omverden* ... med sikte på å forbedre vårt *arkiv* der man lagrer info/data om omverdenen i en modell.**

Utg.pkt.:	**Objekter (entiteter)** **Data**

Begynner med en modell av vår omverden (den del som er interessant).

EAR	**Hvilke entiteter er interessante og hvilken info/data vil man lagre om dem.**
UML	**Hvilke objekter er interessante, hvilken info/data vil man lagre om dem, Hvilke tilstander gjennomgår de og hvordan skal de agere (hvilke "tjenester" skal de kunne utføre)?**

Modell/modellering

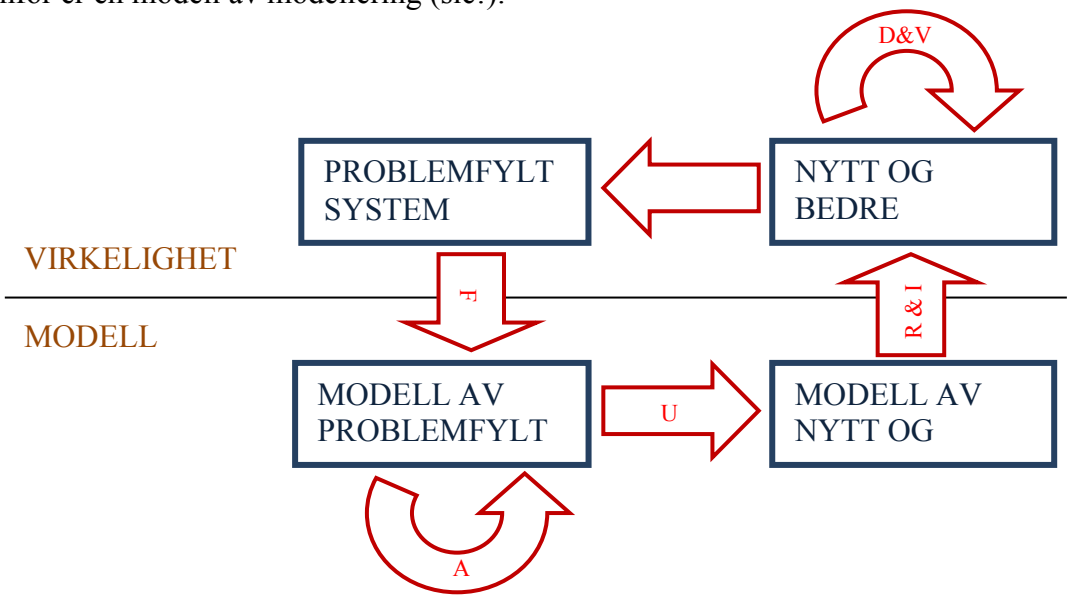
Tenk igjennom: Hva kjennetegner en *modell*? Her er noen modeller:

- ✓ En modelljernbane
- ✓ Et modellfly
- ✓ En matematisk formel
- ✓ En molekylmodell
- ✓ En plastfigur av en skrekkdinosaur
- ✓ En dukke
- ✓ En prototype
- ✓ En verden i et dataspill
- ✓ En datamodell

Hva har disse felles? De er avbildninger av virkeligheten. De kan være forenklet på mange måter: Forminsket, forstørret, abstrakte, mindre detaljer (abstrahert) har ingen metabolisme (de er døde ting) osv.

Man bruker svært mye modeller i systemering og dere har allerede lære flere forskjellige.

Nedenfor er en modell av modellering (sic!).



Forandringsanalyse (avgrense, modellere)

Analyse (studere i detalj, dele opp)

Utforming = Design (lage ny modell)

Realisering (virkeliggjøre)

Implementering (ta i bruk)

Drift (styre systemet)

Vedlikehold (endre for å forandre, legge til funksjonalitet og rette feil)

= FAURID/V (System Development Life Cycle, SDLC)

Etter en tids vedlikehold og turbulens i omgivelsene, blir systemet problemfylt igjen og prosessen må gjentas.

For noen er modellering ekstremt viktig. Her er en annonse fra Phillips Petroleum:

Forestillinger om en virkelighet.

Modellbygging er en form for høyttenkning, d.v.s. en måte å finne sannhet på uten konsekvenser.

Barns lek er en form for modellbruk. Både leketøy og selve leken er etterligninger av virkeligheten slik barn oppfatter den. Barn formes og lærer av å leke.

Modellbruk inneholder alltid et element av lek – en slags barnlig nysgjerrighet som er nødvendig for alle, og som ofte bringer oss ny kunnskap.

Alle virkelige situasjoner er kompliserte fordi mange forhold påvirker dem. En enkel måte å studere og beskrive slike situasjoner på, er nettopp gjennom bruk av modeller.

Modeller representerer en forenkling og behøver ikke ha noen ytre likhet med den virkelighet de skal gjenskape – bare de utfra definerte betingelser oppfører seg som den.

I Norge brukes modeller basert på avansert datateknikk til forskjellige formål. For eksempel brukes en slik modell til simulering av vår samfunnsøkonomi i årene som kommer. På denne måten kan myndighetene få utprøvd og justert sine tiltak før de gjennomføres i full skala.

I andre sammenhenger er betingelsene for modellen gitt. – Spillet Monopol er et godt eksempel. Det samme er tilfelle med en brannøvelse eller bruk av en flysimulator.

Modeller gjør det som er vanskelig enklere å forstå og dermed ofte sikrere.

Også innenfor oljevirksomheten er bruk av modeller fullt ut beretiget. Bygging av installasjoner offshore har mange likhetstrekk med et puslespill. Alle elementer og moduler er innbyrdes avhengige av hverandre og nødvendige for å få den tilsiktede helhet og funksjonsdyktighet.

Modellene tjener som verktøy både for designere, ingeniører, teknikere og montører. Alle som er med får anledning til å sette seg inn i situasjonen og byggverket under kontrollerte og forenklede forhold før produksjons- og monteringsfasen i full skala offshore.

Kostnadene ved slik modellbygging kan løpe opp i mange millioner pr. enhet. Uten tvil mange penger, men relativt ubetydelige i forhold til installasjonenes byggesum. På Ekofiskfeltet er det nå investert ca. 40 milliarder kroner. Modellkostnadene utgjør en svært beskjeden del av dette, men har bidratt vesentlig til at hvert enkelt element fungerer som det skal, at krav om sikkerhet er innfridd og at kostnadsnivået hele tiden har vært under kontroll.

Phillips Petroleum Company Norway

P. O. Box 220 – 4056 TANANGER

"Gode systemer"

Tenk igjennom selv: Hva er et "godt system"? Her er noen tanker om det:

"Gode systemer"

1. er nyttige (useful & usable)
2. kan man stole på (reliable)

3. er fleksible (flexible)
4. har man råd til (affordable)
5. er tilgjengelige (available) for vårt systemmiljø

Opp gjennom tidene har det vært noen dramatiske feilsatsninger på IT[3]:

1. *Ariane 5* – raketten måtte sprenges på vei opp pga programvarefeil. Den direkte årsaken var at et 64 bits reelt tall med fortegn ble konvertert til 16 bits heltall og ga (i visse situasjoner) overflow. Det var én setning uten kontroll for størrelse, men i realiteten skyldtes feilen en lang rekke faktorer herunder testing med forenklede data.
2. *Taurus* – London Stock Exchange var estimert til £6m, prosjektert til £75m og kunden regnet med £450m. Oppgitt. LSE måtte betale over £500m i erstatninger. Senere utviklet de *TradElect* for $68m i fire år, men måtte bytte det ut etter bare to års bruk.
3. *Denver flyplass bagasjesystem* - $200m måtte "repareres" for $100m mens flyplassen sto ferdig uten å kunne brukes i 16 måneder. Bare en liten del ble implementert (resten ble gjort manuelt) og systemet ble kastet i 2005, da manuelt arbeid viste seg billigere. Total ekstra kostnad $560 m.
4. *Therac 35* – radiologisk stråleutstyr ga minst 25 pasienter overdose og tre døde av det.
5. *British Airways/British Airport Authority* – et bagasjesystem til $32m virket ikke. 28000 bagasjeenheter ble mistet. BA måte i en uke nekte passasjerer som hadde bagasje med seg å sjekke inn. BA sjefen oppsummerte det slik: "Not our finest hour!"
6. *Engelske barnebidrag* – Systemet overbetalte og da skattevesenet ville ha pengene tilbake skapte det alvorlige økonomiske problemer for mange og £3 milliarder måtte oppgis.

Å dele opp systemet øker betydelig sjansene for å lykkes. Det er sterkt begrenset hva man kan overblikke (7±2 = "Hrair limit"). Delene man lager kan være bl.a.

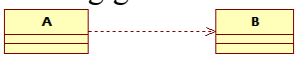
- ✓ Filer
- ✓ Subrutiner
- ✓ Biblioteker
- ✓ Klasser
- ✓ Pakker
- ✓ Uavhengige programmer

Sammenhengen mellom modulene karakteriseres av

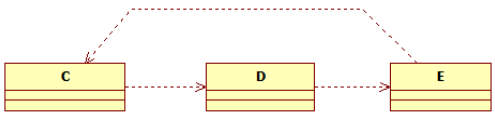
1) Dependency (avhengighet)
2) Coupling (kobling)
3) Cohesion (liming)
4) Interface (grensesnitt)
5) Encapsulation (innkapsling)
6) Abstraction (abstrahering)

Ad 1 – Dependency (avhengighet)

Hvis A må endres når B endres, så er A avhengig av B:



Man bør ha minst mulig avhengighet og absolutt unngå refleksiv avhengighet:

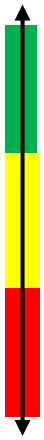


[3] En artig side for dramatiske prosjektfeil er http://calleam.com/WTPF/

Ad 2 – Coupling (kobling)

Kobling er en form for avhengighet. Tradisjonelt – i ren imperativ programmering – var kobling navn på *kall* (modul A kaller modul B). Koblinger er man avhengig av, men de bør være "normale"[4]. Patologisk kobling er også mulig i OOP, f.eks. når objekt *b* feiler og da returnerer en verdi til objekt *a* som den vet vil få *a* til å feile.

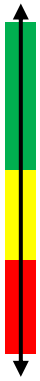
I objekt a: *y=5/b.z()*
I objekt b: *z() feiler og lar da z() returnere 0 for å få a til å feile.*



Ingen:	Modulene har ingen sammenheng med hverandre.
Data:	Alle parametrene er atomære.
Post:	En eller flere av parametrene er post(er) – de er altså da ikke atomære.
Kontroll:	En eller flere av inn-parametrene er kontrolldata
Tramp data:	En eller flere av parametrene er bare "på gjennomreise" for å bli videresendt til en annen modul.
Bundled post:	En eller flere parametrene er en kunstig post der dataene er uten annen sammenheng enn at de skal oversendes samtidig.
Global:	Koblingen skjer via en "global" variabel.
Hybrid:	En eller flere av parametrene har *flere* domener.
Patologisk:	Den kallende modul refererer til noe inne i den kalte (f.eks. endrer koden, endrer data eller hopper inn i en modul).

Ad 3 – Cohesion (liming)

En modul (et objekt i OOP) har høy grad av liming hvis den – konseptuelt – gjør én og bare én ting.



Funksjonell:	Alle elementer bidrar til å få bare én ting gjort (modulens navn)
Sekvensiell:	Output fra en gruppe elementer brukes som input til neste gruppe. Rekkefølgen er avgjørende.
Kommunikativ:	Alle elementer bruker samme data. Rekkefølgen er uvesentlig.
Flytmessig:	En tilfeldig gruppe elementer, men rekkefølgen er vesentlig.
Tidsmessig:	Elementene gjøres på samme tid, men rekkefølgen er uvesentlig.
Logisk:	En bag[5] med elementer som likner på hverandre – flagg avgjør hvilke elementer som brukes
Tilfeldig:	En bag urelaterte elementer, flagg avgjør hvilke som brukes.

Ad 4 – Interface (grensesnitt)

Objekters grensesnitt er det de viser frem til andre objekter. Man bør ha stor kontroll over grensesnittet – i praksis lager man grensesnitt med *kun* funksjoner/metoder, og man lager bare de *nødvendige* funksjonene/metodene "public". Hensikten er å sikre at alle endringer i objektets tilstand skjer kontrollert og da bare av objekter som man ønsker skal få lov til det.

Ad 5 og 6) – Encapsulation og Abstraction (innkapsling og abstraksjon)

Abstraksjon er når klienten ikke *behøver* å vite mer enn det som grensesnittet viser frem. *Innkapsling* er når klienten ikke *kan* vite mer enn grensesnittet.

---

[4] Koblingen sies å være *normal* hvis all informasjon utveksles åpent mellom modulene som parametre. Global og hybrid er da ikke normale i denne forstand.

[5] En *mengde* er et antall elementer av samme type, der ingen er like og det ikke er noen ordning. En *bag* er en mengde som kan ha flere like.

I praksis gjennomfører man begge samtidig ved å privatisere alt unntatt noen public funksjoner/metoder. Da kan ikke klienten vite

- ✓ Hvordan data i tjeneren er lagret
- ✓ Hvordan tjeneren gjennomfører handlingen (programkoden)
- ✓ Om tjeneren har lagret dataene i det hele tatt

F.eks. *varelinje.beløp()* returnerer en *double*.

- ✓ Er beløpet lagret?
- ✓ Er antall og pris lagret?
- ✓ Er bare antall lagret (pris hentes fra et annet objekt)?
- ✓ Hvis antall faktisk er lagret, er det da en streng, eller double, eller int eller…?

OBS! Innkapsling brukes om to forhold:

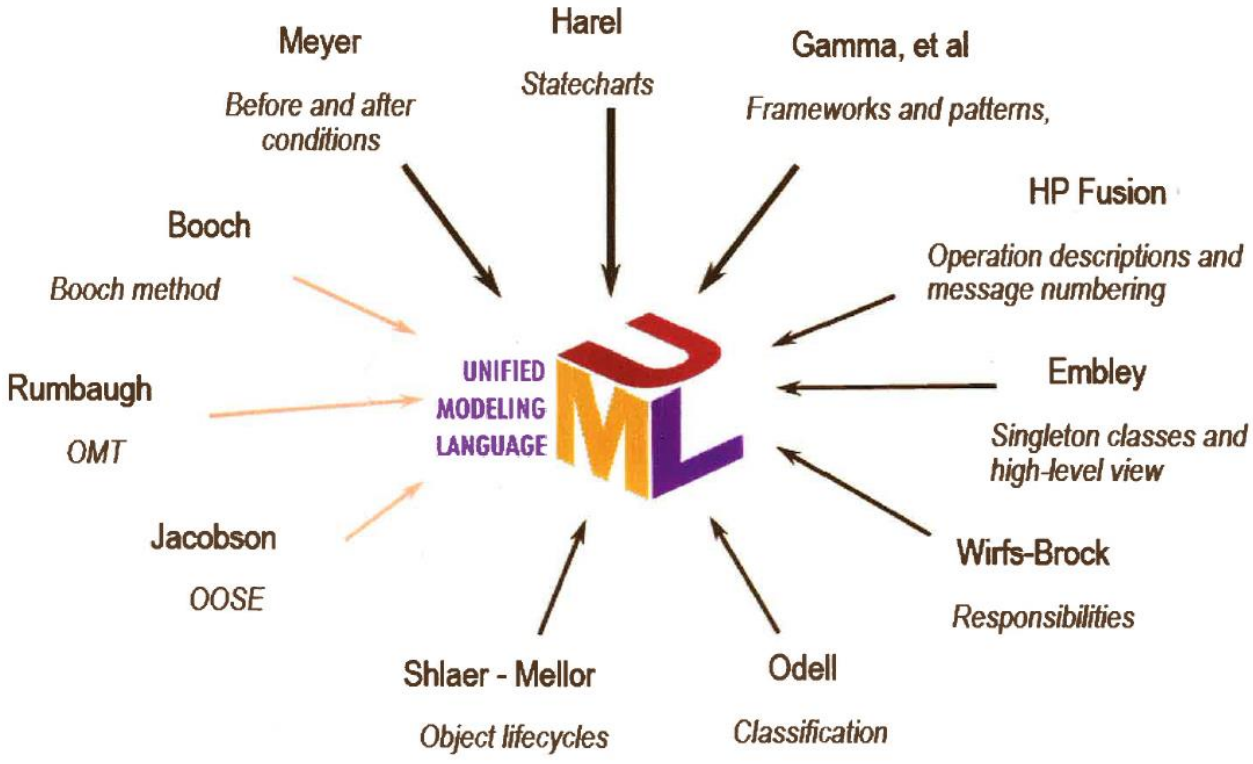
- ✓ At data og funksjonalitet er innkapslet i samme enhet (objektet), og
- ✓ Implementering skjules

Generelt om UML

UMLs kilder

I 1950- og 1960-årene ble det foreslått svært mange teknikker og metoder med tilhørende grafer for analyse og design av IT-systemer. "Alle" ville at deres metode skulle vinne frem, de skrev bøker og de foreleste. Dette er beskrevet som "the great method war".

"The Three Amigos" – Grady Booch, James Rumbaugh og Ivar Jacobsen – slo seg da sammen og laget UML, bygget på sine egne metodeforslag. Første versjon 0.9 var klar i 1996 og versjon 1.0 kom i 1997. De hentet begreper, teknikker og symboler fra svært mange steder. Figuren gir en idé om kildene:



Kilde: http://www.sa-depot.com/?page_id=217

UML og sammenhengen med generell systemteori

Fra generell systemteori, vet man altså at systemer består av deler med sammenhenger mellom. Systemer har også en grense, og utenfor systemet finner man dets omgivelser, bestående av andre systemer som vårt system har relasjon til.

UML deler tilsvarende spesifikasjonen inn i "ting" og "relasjoner". "Ting" er systemdeler, f.eks. handlingsmønstre, klasser, objekter og komponenter. "Relasjoner" er sammenhenger mellom delene, f.eks. "anvender", "sender melding til" og "samarbeider med".

Diagrammene i UML viser systemet sett fra forskjellige sider, kalt "views" – jeg kaller det perspektiver. I de forskjellige perspektivene er det bestemte sider ved systemet som er i fokus. Avhengig av hva som er fokus i perspektivet, representerer delene forskjellige "ting" og relasjonene forskjellige slags sammenhenger.

Analogt kan en arkitekt tegne forskjellige slags diagrammer:

- ✓ Plantegninger viser rominndelingen. Delene er her rom og sammenhengen vises ved hvordan rommene ligger inntil hverandre, dører, trapper o.l.
- ✓ Elektriske tegninger viser komponentene i det elektriske anlegget, og sammenhengene viser ledningen der strømmen skal gå fra komponent til komponent
- ✓ I rørleggertegningene er vask, badekar, varmtvannsbeholder osv. deler og sammenhengen mellom dem er rør der vannet skal renne i en bestemt retning
- ✓ Osv.

Når man tegner datasystemer, kan delene være f.eks. funksjonalitet, klasser, moduler og maskiner, avhengig av perspektivet. Hva slags relasjoner det er mellom dem, avhenger av hva slags deler det er snakk om. F.eks. kan det være arv mellom klasser, meldinger mellom objekter og det kan være ledninger med Ethernet mellom maskiner.

For å skille typer av "ting" og typer av "relasjoner", bruker UML forskjellige symboler. F.eks. tegnes en klasse forskjellig fra en maskin, og arv tegnes annerledes enn en melding. *Symbolenes betydning utgjør UMLs semantikk.*

Symbolene kan settes sammen på mange forskjellige måter, men noe er lovlig, annet er det ikke. *Reglene for hvordan symbolene kan settes sammen, utgjør UMLs syntaks.*

Regler for hvordan ord bøyes i kjønn, tall, tid osv. kalles et språks grammatikk, men kunstige språk som UML har sjelden grammatiske regler. *Det er ingen grammatikk i UML*[6].

Definisjon:
The Unified Modeling Language (UML) is a language for specifying, visualizing, constructing, and documenting the artifacts of software systems, as well as for business modeling and other non-software systems. The UML represents a collection of the best engineering practices that have proven successful in the modeling of large and complex systems.

Kilde: UML 1.4, Foreword p. xix

[6] Det finnes riktignok ett eksempel der et eget symbol angir "flere objekter" og det kan kanskje tolkes som flertallsbøyning.

Om UMLs teknikker (modeller/grafer):

The choice of what models and diagrams one creates has a profound influence upon how a problem is attacked and how a corresponding solution is shaped. Abstraction, the focus on relevant details while ignoring others, is a key to learning and communicating.

...

In terms of the views of a model, the UML defines the following graphical diagrams:

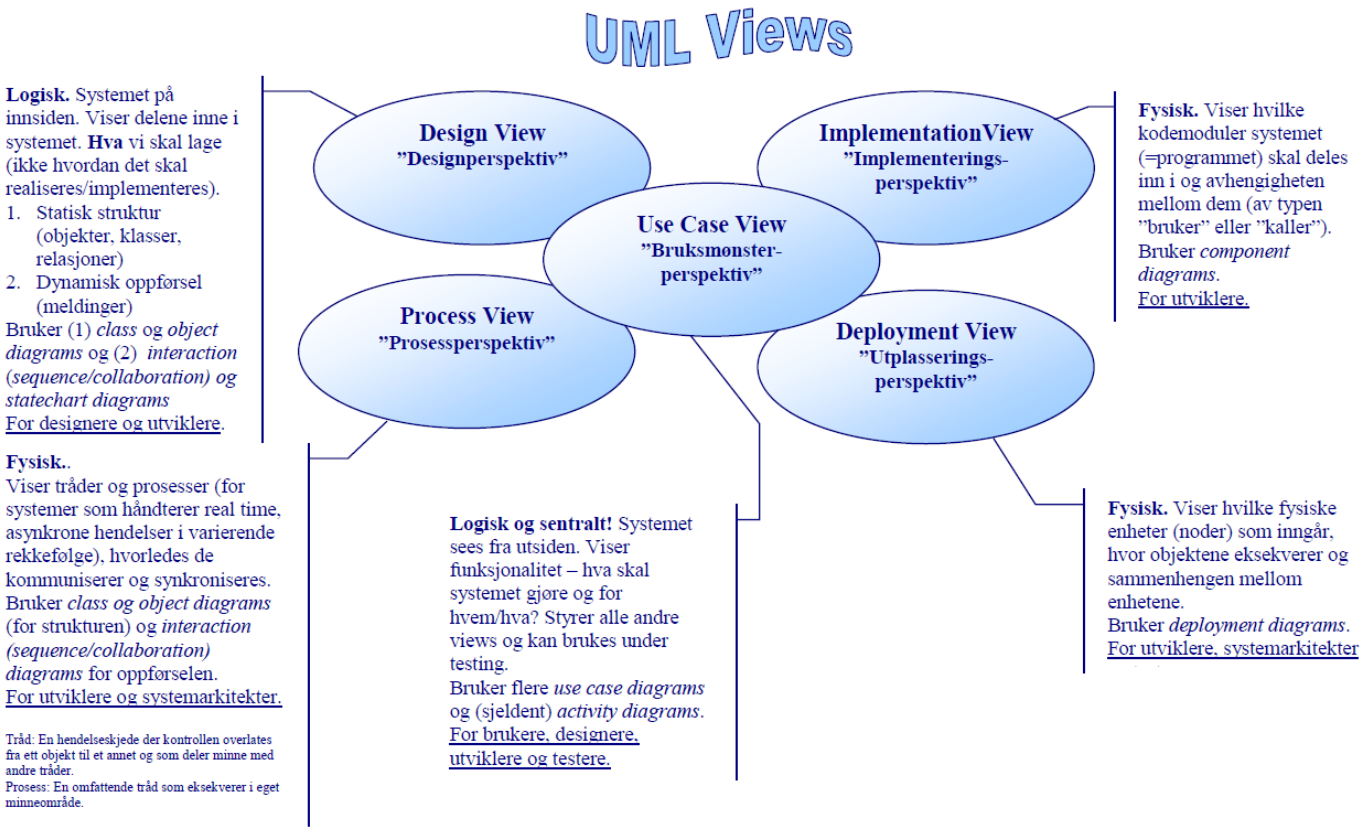
- ✓ *use case diagram (bruksmønsterdiagram)*
- ✓ *class diagram (klassediagram)*
- ✓ *behavior diagrams (oppførselsdiagrammer):*
  - *statechart diagram (tilstandsdiagram)*
  - *activity diagram (aktivitetsdiagram)*
- ✓ *interaction diagrams (samhandlingsdiagrammer):*
  - *sequence diagram (sekvensdiagram)*
  - *collaboration diagram (samarbeidsdiagram)*
- ✓ *implementation diagrams (implementeringsdiagrammer):*
  - *component diagram (moduldiagram?)*
  - *deployment diagram (utplasseringsdiagram?)*

Although other names are sometimes given to these diagrams, this list constitutes the canonical diagram names.

Kilde: UML 1.4, Summary, p. 1-2, forslag til oversettelser i parentes)

UMLs perspektiver (views)

Med referanse til omtalen ovenfor av perspektiver i systemering, ser UML systemet fra fem, forskjellige sider som de kaller "views". Tilsammen skal de altså danne et komplett bilde av systemet. Hvert perspektiv har sine egne beskrivelsesteknikker, herunder diagrammer. Nedenfor har jeg laget en figur som viser perspektivene og hvilket synspunkt de representerer, samt diagrammene som benyttes.

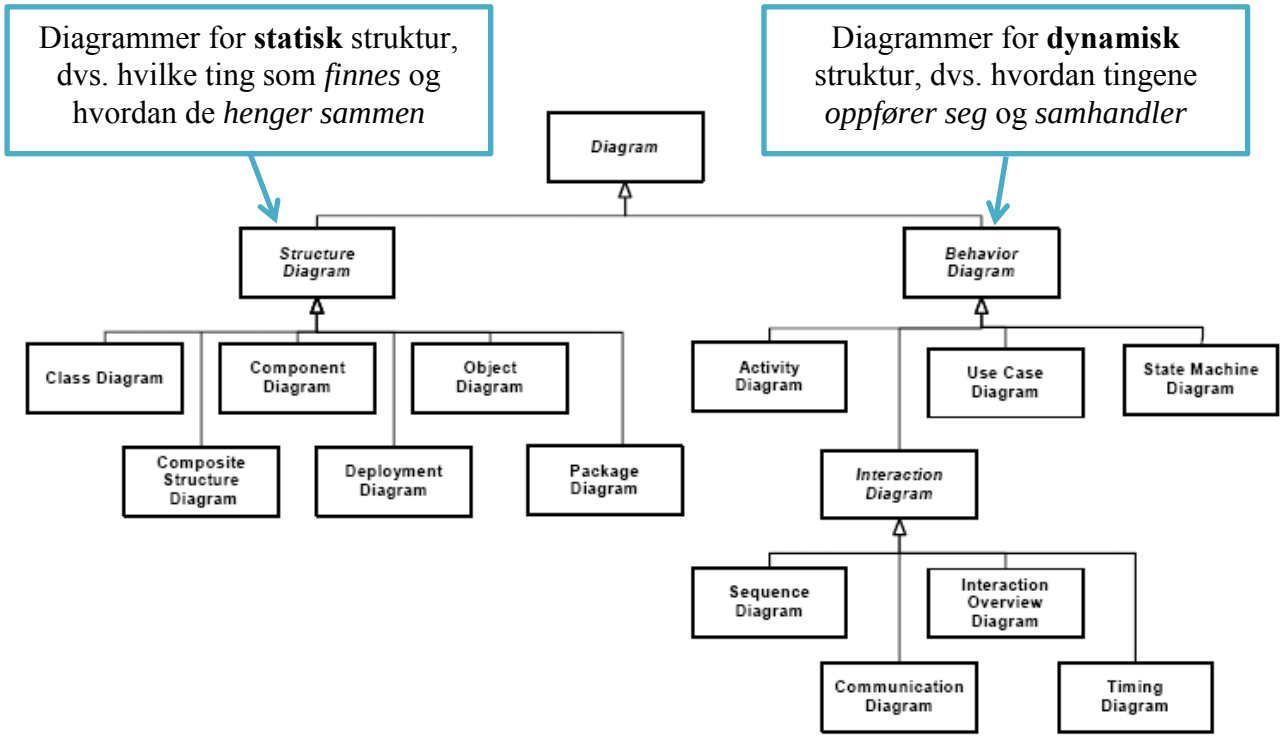


Alle perspektivene er på samme "nivå" – det er ikke angitt at noen av dem er viktigere enn andre. Det vil også variere med typen av system hvilket perspektiv det må arbeides mest med og som må dokumenteres mest grundig. Det er allikevel vanlig å tenke seg at arbeidet bør starte med bruksmønsterperspektivet der systemet ses fra utsiden, da det dokumenterer de funksjonelle systemkravene. Det følger jo av det teleologiske prinsippet (se ovenfor) at systemet skal bedømmes etter sitt formål. Formålet dokumenteres i bruksmønsterperspektivet og derfor ser jeg dette perspektivet som sentralt.

Der er også anledning til å arbeide litt med ett perspektiv, så litt med et annet for deretter å vende tilbake til det første for mer arbeid med det.

Oversikt over diagrammer

Denne figuren er hentet fra standarden og viser hvilke diagrammer som brukes i UML og hvordan de henger sammen:



Kilde: "Unified Modeling Language: Superstructure", versjon 2.0

Kritikk av UML

UML består av hele *ti* diagramteknikker, men i realiteten er det enda flere pga varianter. I tillegg kommer andre spesifikasjoner som DDL, tekster o.l. Det er sammenhenger mellom alle teknikkene, og man må passe på at spesifikasjonen er indre konsistent. Det er altså ikke enkelt å lære seg alt dette. For en ”vanlig *bruker*” er det i virkeligheten umulig (en bruker som kan alt dette er dessuten ikke lenger en ”vanlig bruker”). Selv i UMLs egen dokumentasjon er ikke alltid sammenhengene mellom dokumentasjonsformene klar.

Dette åpner for *modellmakt* ved at IT-ekspertene anvender et språk og modeller som brukerne ikke har forutsetninger for å forstå. Brukernes innflytelse blir derved sterkt redusert, hvilket er problematisk i skandinavisk sammenheng. Jeg tror egentlig ikke det er realistisk at brukerne er med på annet enn bruksmønstrene, deretter ”etterlates de i støvet”. Heldigvis er bruksmønsterperspektivet svært sentralt i UML, men likevel vil mange beslutninger som tas senere påvirke systemet sterkt – også på måter som brukerne merker.

Det kan være problematisk at hele analysen forutsetter at systemet skal realiseres med edb, og at det skal være objektorientert. Det er slett ikke sikkert at det er den beste, eller eneste, løsningen. Kanskje skal bare deler av systemet realiseres med edb, kanskje bare deler av det igjen med OOP. Når systemet først er analysert objektorientert, er det imidlertid ikke alltid trivielt å realisere det med andre edb-verktøy. Det blir omtrent som å bygge et hus i betong, når det er tegnet i tre: Umulig er det ikke, men det krever betydelige tilpasninger og kanskje noe mer dokumentasjon underveis. F.eks. vet man at de fleste realiserer datalagringen med relasjonsdatabase – allikevel forutsetter UMLs logiske perspektiv at det skal benyttes en objektorientert database.

Det er en ulempe at det ikke finnes noen ”offisiell” metode knyttet til UML. Det er ikke tilstrekkelig å kjenne et stort antall diagramteknikker – det gjorde man fra før – man bør også få vite hvordan de er tenkt brukt. Hvilken rekkefølge jobber man i, hvem deltar osv.? Bare med en slik metode er det f.eks. mulig å planlegge prosjektet. Fordelen med en ”offisiell” metode, er at den utvikles gjennom forskning og erfaringsutveksling. Rationals metode RUP tilfredsstiller ikke kravene til åpenhet.

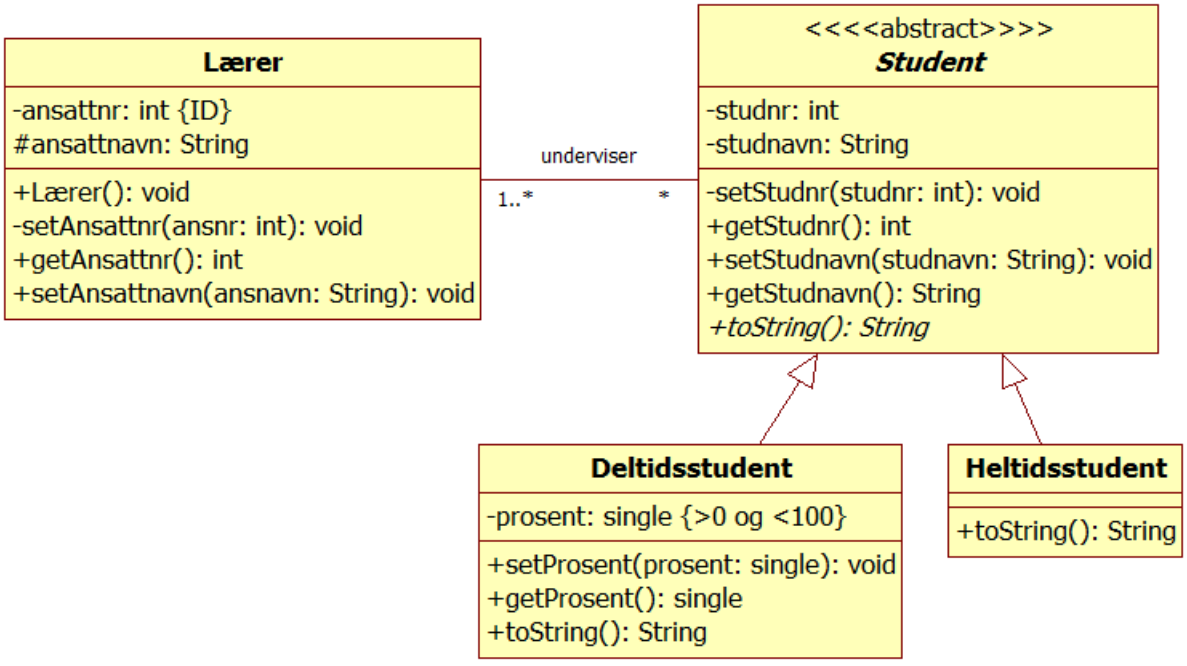
Når UML likevel er en ”vinner” – for det er den – skyldes det mest de tunge aktørene som står bak, med stor markedsmakt. Dessuten var mange lei av ”the Method War”. Videre er det en fordel for adapsjonen at UML er omfattende (med dokumentasjonsformer for det meste) og fleksibel. Man kan i stor grad bruke diagrammene som man vil og legge til egne notasjonsstandarder. Dermed står man sjelden fast når noe skal dokumenteres.

UML er virkelig objektorientert. Mange teknikker og metode som ble lansert under ”the Method War” kalte seg objektorienterte, man var det egentlig ikke. Problemet var at ordet ”objekt” bare betyr ”ting” og *alle* kunne jo hevde at de analyserte ”ting”. Objektorientert som i ”Objektorientert Programmering” var det slett ikke alle som var, eller de tok bare for seg *noen* av de nødvendige perspektivene.

Oppgave til kapittel 1 (StarUML)

Oppgave A: Last ned StarUML (http://staruml.sourceforge.net/en/) og installer den.

Oppgave B: Sjekk at StarUML virker ved å legge inn nedenstående diagram, kalt "Lærerstudent".



Merk at klassen *Student* er abstrakt og det er også metoden *toString()* i denne klassen.

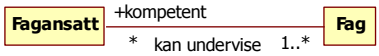
Kapittel 2 – Generelle mekanismer: Noter, stereotyper og relasjoner

Generelle mekanismer

I beskrivelsene som lages innen hvert perspektiv, kan man benytte mange forskjellige mekanismer. Noen mekanismer er generelle og kan benyttes i alle diagrammer. Her beskriver jeg dem.

Kommentarer

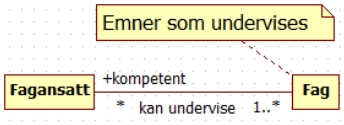
Kommentarer (*annotations*) er ren tekst som står inntil et symbol for å forklare det nærmere. Den utgjør en spesifikasjon av symbolet, men endrer ikke betydningen av det.



I ovenstående figur er en relasjon beskrevet som "kan undervise" og det ene endepunktet er beskrevet som "kompetent". Dette klargjør modellen, men endrer ikke betydningen hverken av streken mellom "Fagansatt" og "Fag" eller rektangelet.

Noter

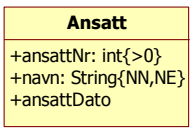
Noter (*notes*) er også ren, beskrivende tekst. De skrives i eget symbol som likner et ark med et hjørne brettet inn. Kommentaren knyttes til det som kommenteres med prikket linje. Det som står i noten endrer ikke noe i modellen, så det er lovlig å skrive hva som helst. Det er vist mange slike i etterfølgende diagrammer.



Her ser man at klassen "Fag" kan beskrives som "Emner som undervises". Symbolet – rektangelet – angir en klasse og det betyr symbolet enten noten står der eller ikke.

Restriksjon/egenskapsstreng

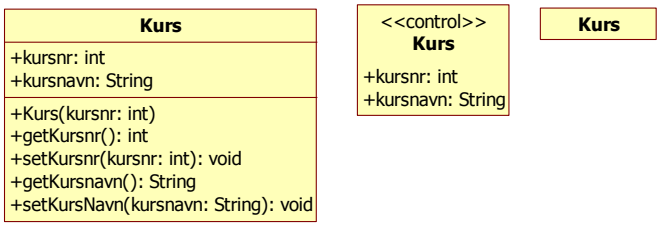
Noen ganger vil man *innskrenke* noe, f.eks. slik at ett attributt skal ha mindre domene enn typen tyder på eller en assosiasjon som det gjelder spesielle regler for.



En ansatt i ovenstående figur skal ha et ansattnummer og det skal være et heltall. I tillegg er det angitt at heltallet skal være større enn null. Videre skal navnet være en streng, men den får ikke være *null* (NN) og ikke en tom streng (NE). Begge er begrensninger – restriksjoner – på det angitte, generelle domenet. Noen få restriksjoner, f.eks. XOR for relasjoner, er forhåndsdefinerte, men ellers bruker man vanlig språk.

Abstrahering

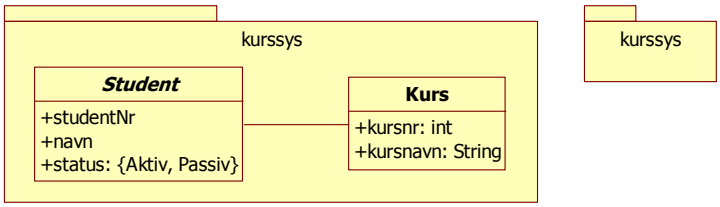
Ofte vil det være klargjørende å *skjule* informasjon i diagrammer fordi det skal vises til noen som ikke er interessert i detaljene. Detaljene blir altså ikke fjernet, bare holdt skjult. I UML er det alltid tillatt, men selvsagt må litt være igjen så det ikke blir meningsløst. Man kan se på det som en alternativ visning. Nedenfor vises tre slike "visninger" av en klasse. Alle har samme betydning og ingenting er fjernet, men det *vises* bare mindre og mindre – resten er abstrahert bort her.



Prinsippet er "need-to-know", altså at man bare viser det som faktisk er interessant i sammenhengen.

Pakker

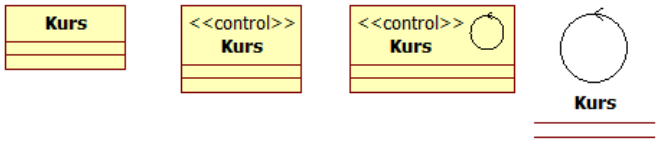
En mekanisme som benyttes mye i OOP er *pakker*. Det innebærer at klasser ligger inne i en samling. I UML benyttes eget symbol til dette, og det kan da også benyttes som en egen abstraksjonsmekanisme. Nedenfor vises pakken "kurssys" først med og deretter uten innhold (innholdet er abstrahert bort).



Stereotyper

Stereotyper benyttes i UML til å lage varianter av vanlige symboler. Stereotypene innebærer en utvidelse av språket. Det tilbys et sett av standard stereotyper, men det er også lov å lage sine egne. Stereotypene skrives inntil eller inne i symbolet som skal endres, omsluttet av << og >> (dette er egentlig «gammeldagse» anførselstegn, kalt «guillemets» vanligvis brukt for å indikere direkte tale). *Stereotyper endrer symbolets betydning.*

Stereotypene kan også ha et ikon tilknyttet seg. Da kan man velge om man vil bruke teksten eller ikonet eller begge deler. Her er f.eks. en klasse *Kurs* tegnet som en helt vanlig klasse, og som en kontrollklasse på tre, forskjellige måter – den siste er ikke særlig vanlig:



Stereotypen <<*control*>> angir at *Kurs* ikke er en helt vanlig klasse, men en variant.

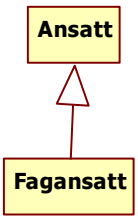
Stereotypene kan knyttes til et hvilket som helst annet symbol, selv om noen tilknytninger er mer vanlige enn andre. De som er gjengitt og definert i UMLs liste over standard stereotyper, brukes naturligvis mest.

Relasjoner

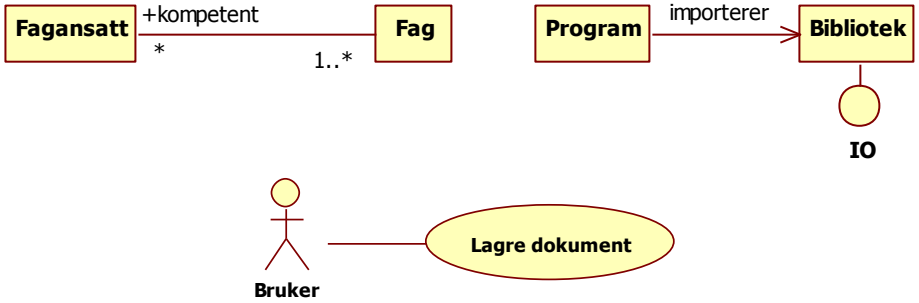
I UML antar man at systemet er et elektronisk datasystem som skal realiseres objektorientert. Man vil beskrive dette systemet fra mange perspektiver, og delene er da av forskjellige slag. Det kan f.eks. være klasser, objekter, omgivelser, pakker, maskiner, komponenter, tilstander eller grensesnitt. Mellom disse – som utgjør delene i forskjellige perspektiver – er det da *relasjoner*. Disse relasjonene vil være av forskjellig type, avhengig av hva slags deler det er snakk om.

De har forskjellige navn, forskjellig notasjon og brukes til å angi forskjellige typer av sammenhenger. Her skal jeg gi en oversikt over disse relasjonstypene.

1) **Arv** (inheritance, generalization).
Arv er omtalt mer i detalj senere i kapitlet.
   a) Kan bare benyttes mellom *to elementer av samme slag*, vanligvis to klasser eller to bruksmønstre, og kan ikke være refleksiv (fra og til samme element som når A arver fra A).
   b) Relasjonen innebærer at den spesialiserte arver *alle* egenskaper til den generelle, men har enten *tillegg* eller *endringer* i forhold til den.
   c) Tegnes med heltrukken strek og lukket pilspiss uten fyll, fra spesialiseringen (sub-) til generaliseringen (meta-).



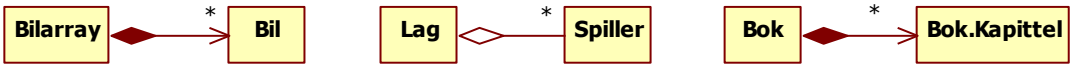
2) **Assosiasjon** (association).
Assosiasjoner er omtalt mer i detalj senere i kapitlet.
   a) Assosiasjon benyttes i mange sammenhenger, herunder mellom to klasser, mellom aktør og bruksmønster, mellom to aktører (sjeldent og på kanten av UML standard), mellom to komponenter, mellom to noder, samt mellom en komponent og et grensesnitt. Assosiasjon kan gå mellom to elementer av forskjellig slag og den kan være refleksiv (A assosieres med A).
   b) Assosiasjon innebærer at de to elementene er bundet sammen på en eller annen måte.
      i) Mellom to klasser, betyr det at objektene i den ene klassen "vet om" ("=kjenner referansen til") ett eller flere objekter i den andre klassen og assosiasjonen sies å være "navigerbar". Hvis assosiasjonen er enveis, er det bare objektene i den ene klassen som kjenner til objektene i den andre, og ikke omvendt.
      ii) Mellom en aktør og et bruksmønster betyr assosiasjonen at aktøren har ønske om og/eller behov for at systemet skal kunne utføre bruksmønsteret. Slike assosiasjoner er aldri pilsatt, men kan ha kardinalitet (semantikken for dette er udefinert).
      iii) Mellom en komponent og et grensesnitt, betyr assosiasjonen at komponenten tilbyr grensesnittet.



   c) **Varianter** av assosiasjoner er

i) **aggregering** = shared aggregation med en åpen rombe som underforstår ”mange” som kardinalitet
ii) **bestanddel** = composition med en fylt rombe som underforstår ”en” som kardinalitet
iii) Tidligere fantes også **indre klasse** (også kalt "nøstet klasse") = nested class der den ene klassen er en indre, lokalt deklarert klasse i forhold til den andre, dvs. formelt innenfor klassens ”navnerom”. Det fantes egen notasjon for dette i tidligere versjoner av UML men det er nå forlatt.

d) Assosiasjon tegnes som en heltrukken linje. Assosiasjonen kan spesifiseres med f.eks. kardinalitet, roller og navn/verb, uten at det endrer betydningen. Den kan også utstyres med en åpen pil i den ene enden, og sies da å være ”enveis”.



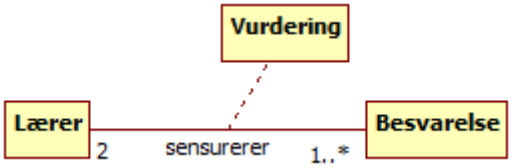
3) **Assosiasjonsklasse** (association class)

a) Assosiasjonsklassen er knyttet til hver enkelt forekomst av en assosiasjon mellom to klasser, dvs. til hver assosiasjon mellom ett objekt i den ene klassen og ett objekt i den andre. Normalt vil assosiasjonsklasser bare tilknyttes assosiasjoner som har kardinalitet mange-til-mange. Formelt arver assosiasjonsklassen egenskapene både til assosiasjon og til klasser, og regnes følgelig som en variant av assosiasjoner.

b) Assosiasjonsklassen spesifiserer assosiasjonen nærmere (gjennom attributter) og kan utføre tjenester vedrørende assosiasjonen (gjennom metoder). Dette er bare en alternativ tegnemåte, slik at en assosiasjon A⇔B med assosiasjonsklasse Z, er det samme som tre klasser A, B og Z med tre assosiasjoner A⇔Z, Z⇔B (som en ”objektering av relasjon” i databasemodellering) og A⇔B.

c) Det er bare ett assosiasjonsobjekt for hver assosiasjon mellom to objekter. Det er underforstått at
i) hvert assosiasjonsobjekt er assosiert med bare ett objekt på hver side som på sin side er assosiert med assosiasjonsobjektet med samme kardinalitet som de har i forhold til den andre klassen (normalt mange)
ii) de to objektene er i tillegg også assosiert direkte (alternativt kan denne assosiasjonen realiseres med metoder i assosiasjonsobjektet)

d) Tegnes med en prikket linje som forbinder assosiasjonsklassen med midten av den assosiasjonen den tilknyttes.



4) **Avhengighet** (dependency)

a) Avhengighet knytter sammen ett element og et annet element. Avhengigheten kan hverken være refleksiv (A→A) eller rekursiv (A→B→A).

b) Avhengighet innebærer at det ene elementet på en eller annen måte er avhengig av det andre. Tegnet på det, er at endringer i det ene elementet kan kreve endringer i det andre.

c) Det finnes flere varianter av avhengighet:

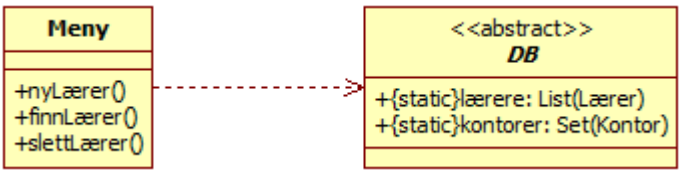
i) I *klassediagrammer* trekkes avhengigheten mellom klasser. Da innebærer avhengigheten at (objekter i) den ene klassen på en eller annen måte *bruker* (objekter i) den andre klassen, dvs sender meldinger til dem.

ii) I *bruksmønsterdiagrammer* trekkes avhengigheten mellom bruksmønstre. Avhengigheten er her alltid merket med en av to stereotyper:
- «Includes» innebærer at det ene bruksmønstre alltid inneholder det andre (hvor det andre bruksmønsteret skal inkluderes = *insertion point*, kan spesifiseres).
- «Extends» innebærer at det ene bruksmønsteret på visse vilkår *kan* benyttes av det andre.

iii) I *komponentdiagrammer* viser avhengigheten at en komponent anvender et grensesnitt, eller en annen komponent.

iv) I noen *andre diagrammer* benyttes avhengigheten til å vise **realisering** (realization), som når et samarbeidsdiagram realiserer et bruksmønster.

d) Avhengighet tegnes som en brutt linje med åpen pil.



5) **Meldinger/kall** (messages/calls)

a) Meldinger og kall brukes i *sekvensdiagrammer* og innebærer at ett objekt/klasse anvender en metode i den andre klassen. Den spesifiseres med metodens grensesnitt.

b) Betydningen av de tre variantene er:

i) **Synkront kall** innebærer at kallende objekt/klasse sender meldingen og venter på svar. Det er underforstått at det gis svar (returverdi), eller i alle fall et signal om at metoden er ferdigeksekvert avslutning (når metoden er *void*).

ii) **Asynkront kall** innebærer at kallende objekt/klasse *ikke* venter på svar, men fortsetter eksekveringen med en gang. Det kalte objektet må da eksekvere i egen tråd (thread) eller prosess (process = en tråd med eget minneområde). Hvis det skal gis svar, sender da utførende objekt/klasse et avbruddssignal.

iii) **Returverdi** er en reaksjon på et tidligere sendt kall. Hvis det er nødvendig følger det med et avbruddssignal, så kallende objekt/klasse kan vite at svaret er klart.

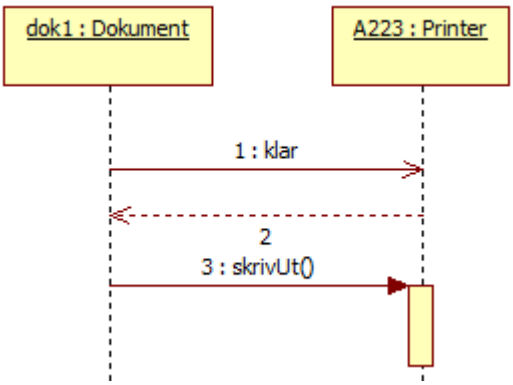
c) I *samarbeidsdiagrammer*, som er tett knyttet opp til sekvensdiagrammer, benyttes assosiasjoner (se ovenfor) som er spesifisert med en pil ved siden av. Pilen angir retningen for kallet, og er ytterligere spesifisert med metodens signatur.

d) Meldinger/kall tegnes som en pil, på tre forskjellige måter, avhengig av hva slags kall det er snakk om:

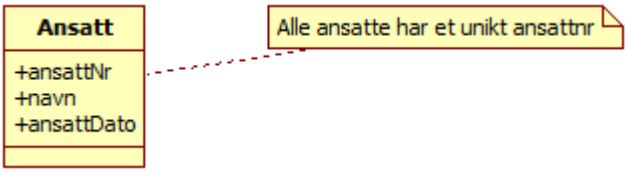
i) Synkront kall: Heltrukken linje med en lukket, fylt pil

ii) Asynkront kall: Heltrukken linje med åpen pil

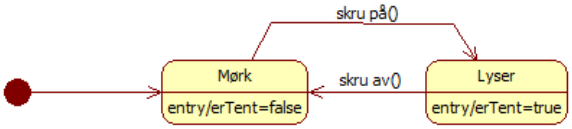
iii) Returverdi: Prikket linje med åpen pil



6) **Notetilknytning** (note attachement)
   a) Tilknytningen går alltid fra en note til noe annet.
   b) Tilknytningen tegnes som en prikket linje, uten pil eller andre spesifikasjoner.



7) **Tilstandsovergang** (state transition)
   a) Tilstandsoverganger brukes bare mellom tilstander i tilstandsdiagrammer. Overgangen kan være refleksiv.
   b) Tilstandsovergangen betyr at objektet, når det mottar visse meldinger eller signaler, går fra én tilstand til en annen. Overgangen er avhengig av hvilken tilstand objektet er i før meldingen kommer. Mens objektet er i en overgangsfase, vil det ikke gjøre noe annet, og kan ha en inkonsistent tilstand.
   c) En variant av tilstandsoverganger finnes i handlingsdiagrammer (action diagram) der tilstandene er *handlinger* eller *subprosedyrer*. Her kan pilene tegnes med brutt linje (asynkron tilstandsovergang der den som sender meldingen ikke venter på at mottaker skifter tilstand).
   d) Tilstandsovergang tegnes som heltrukken strek med åpen pil i den ene enden. Den kan spesifiseres med en *stimulus* (en melding/metodekall på objektet) og en *respons* (i form av sending av meldinger/signaler og sideeffekter, f.eks. endring av ytre lager eller fjerning av objektet).



8) **Flyt** (flow)
   a) Flyt viser et objekt over tid, med stereotypene
      i) «becomes» (objektet endrer klasse)
      ii) «copy» (objektet blir en kopi av et annet – en klone)
   b) Flyt tegnes med en brutt linje med åpen pil og en stereotype.

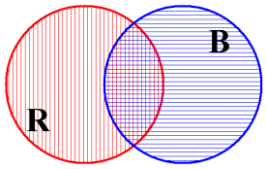
Mer om arv

Generelt

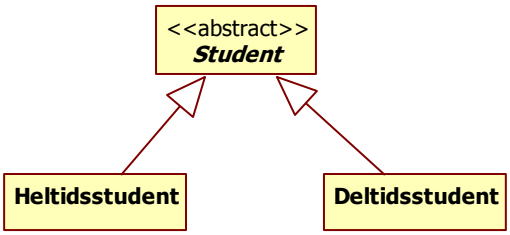
Arv gjør det mulig å deklarere/definere[7] det som er felles bare ett sted. Det har klare fordeler under testing og vedlikehold. Arv er derfor aktuelt hvis klasser har noen felles egenskaper, nemlig like medlemmer[8] (attributter og metoder).

Normalt vil både deklarasjonen og definisjonen av medlemmet være like. Arv gjør det imidlertid også mulig å behandle klasser polymorft. Da har klassene like deklarasjoner, men forskjellige definisjoner. Polymorfismen gir også klare programmeringsfordeler.

Situasjon A: To klasser har noen like egenskaper



Klassene R og B har felles egenskaper, men begge har også egenskaper som den andre klassen ikke har. Det er aktuelt med et arvehierarki der de felles egenskapene legges til en abstrakt metaklasse:



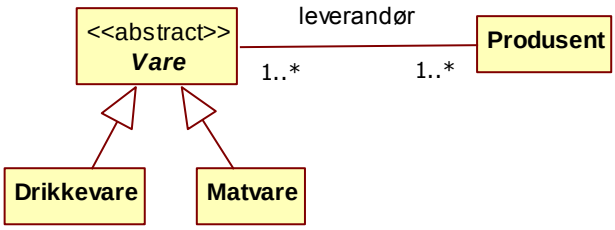
Metaklassen kan ikke instansieres siden den er abstrakt, men alle dens medlemmer arves av de to studenttypene og lagres som en del av dem. *Heltidsstudent* og *Deltidsstudent* er endret i forhold til *Student*, med tillegg/overstyring og andre relasjoner. (Det er regler i programmeringsspråket for hva slags endringer som er tillatt.)

Det er ikke alltid like lett å se at klassene har noe felles:

Her har både Drikkevare og Matvare en relasjon (assosiasjon) til Produsent, og relasjonene er tydeligvis like. Da er arv aktuelt:

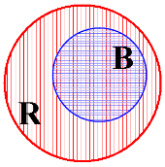
[7] Deklarasjonen gjelder navn, datatype, domene for attributter og hele signaturen for metoder. Definisjonen gjelder implementeringen av metoder (programkoden).

[8] Hvis klassene har felles relasjoner, dvs assosiasjoner, aggregering, avhengigheter og arv, vil det gjenspeile seg i felles medlemmer, nemlig de attributtene som peker til relaterte objekter og metodene som håndterer relasjonen.

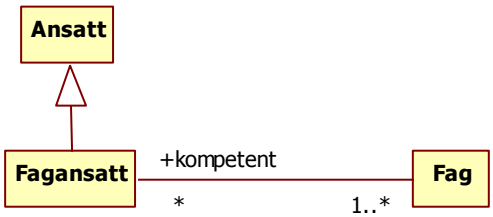


Både drikkevare og matvare arver assosiasjonen til Produsent. Fordelen er både at assosiasjonen nå kan realiseres bare én gang, men også at *Produsent* kan behandle *Vare* polymorft.

Situasjon B: En klasses egenskaper er inkludert i en annen klasses

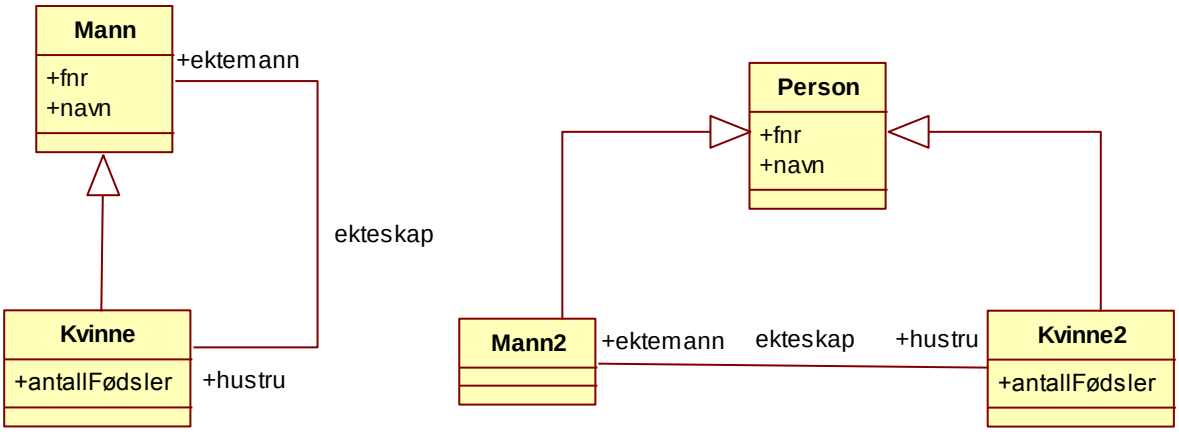


Klassen B har alle sine egenskaper felles med klasse R, som imidlertid har egenskaper i tillegg som ikke B har.



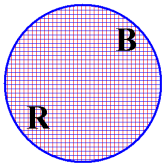
Alle ansatte har visse egenskaper, men fagansatte har en relasjon (assosiasjon) i tillegg (som nedfelles i et attributt).

Noen ganger kan det se ut som situasjonen er av type B, mens den faktisk er av type A:



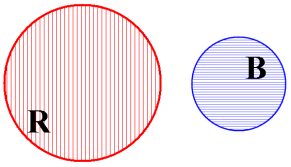
Hvis man tenker at "*Kvinne2* er som *Mann2*, men med noe ekstra" – dvs. situasjon B - tegner man som til venstre. Det blir feil, fordi *Kvinne2* da arver *Mann2*s assosiasjon til *Kvinne2*, slik at en kvinne arver muligheten til å ha rollen som "ektemann" til en kvinne. Situasjonen er altså som i A og man må innføre en klasse *Person* og tegne som til høyre.

Situasjon C: Klassene har alle egenskapene like



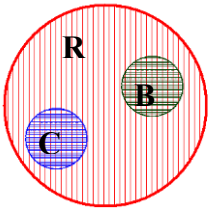
De to klassene er altså helt like. Da er de i virkeligheten ikke to, forskjellige klasser, og de bør slås sammen til én klasse. Arv er da ikke lenger aktuelt.

Situasjon D: Klassene har ingen like egenskaper

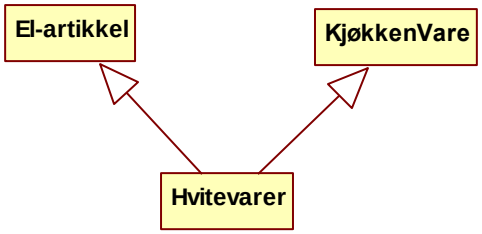


Her er det ikke aktuelt med arv – det er ingenting å arve.

Situasjon E (variant av B): To klassers egenskaper er inkludert i en annen klasses



Klassene B og C har alle sine egenskaper felles med klasse R, som imidlertid har egenskaper i tillegg som hverken B eller C har. Dette kan gjennomføres med *multippel arv* hvis OOP-språket støtter det (som f.eks. C++):



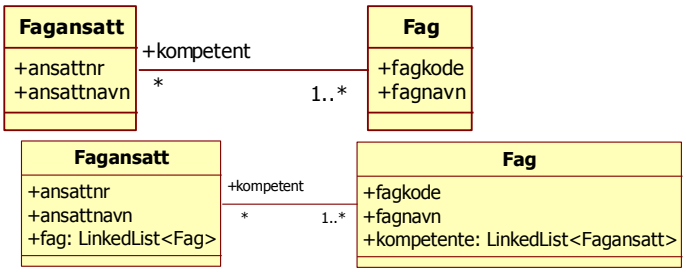
Alle el-artikler har visse egenskaper. Det samme gjelder alle kjøkkenvarer. Hvitevarer (komfyr, kjøleskap osv.) har egenskaper både som el-artikler og som kjøkkenvarer, og i tillegg visse egenskaper som bare hvitevarer har.

Hvis OOP-språket ikke støtter multippel arv (f.eks. Java), kan situasjonen *ikke* løses ved arv – man må ty til andre teknikker som faller utenfor dette notatet.

Mer om assosiasjoner

Assosiasjon mellom to klasser

Assosiasjoner er *navigerbare*. Man kan tenke på det som at objektene i den ene klassen "vet om" objektene i den andre. Det vil i realiteten si at de har et attributt som refererer til de andre objektene. Det er ganske vanlig å forenkle tegningen ved ikke å ta med disse attributtene da de er underforståtte.

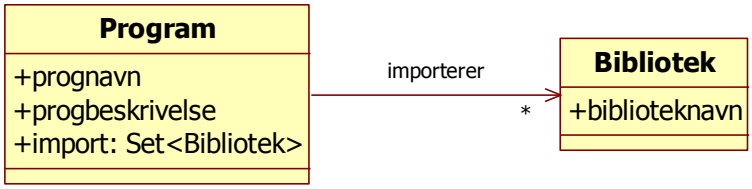


I diagrammet til venstre er attributtene som realiserer assosiasjonen underforståtte. I diagrammet til høyre er de tatt med som *assosiasjonsattributter*. I prinsippet er diagrammene like, men man har jo presisert litt mer til høyre (hva slags samling referansene skal utgjøre og hva den skal hete) – til venstre overlater man det til programmereren. Programmereren vil uansett skrive omtrent noe slikt (Java):

```
import java.util.*;
public class Fagansatt
{
    private int ansattnr;
    private String ansattnavn;
    private LinkedList<Fag> fag = new LinkedList<Fag>();
}
public class Fag
{
    private String fagkode;
    private String fagnavn;
    private LinkedList<Fagansatt> kompetente = new LinkedList<Fagansatt>();
}
```

Listen *kompetente* i et fag-objekt gjør det mulig å finne de fagansatte som er kompetente for et visst fag-objekt som man har tak i. Det er det som mer konkret menes med at assosiasjonene er *navigerbar* fra *Fag* til *Fagansatt*.

Det er vanlig at assosiasjoner er navigerbare begge veier – de sies da å være "toveis" (motsatt "enveis"). Hvis assosiasjonen bare er enveis, må den gjøres retningsbestemt med pil i en ende. Da vil det bare bli assosiasjonsattributt i den ene av klassene:

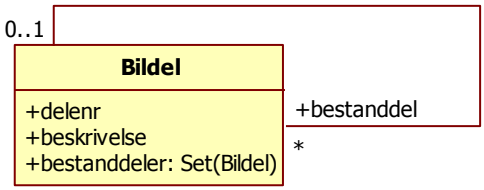


Her har programmene en mengde med referanser til de bibliotekobjektene som den importerer, men det holdes ikke orden på hvilke programmer et gitt bibliotek importeres til.

Assosiasjonene kan utstyres med diverse kommentarer (annotations). Her er kardinalitet (multiplicity), navn og rolle (med synlighet) som objektene spiller i assosiasjonen ført på:

Jeg minner om at slike kommentarer *ikke* endrer betydningen av assosiasjonen.

Assosiasjoner kan være *refleksive* og *rekursive*[9]. Her består f.eks. bildeler av 0..* andre bildeler:

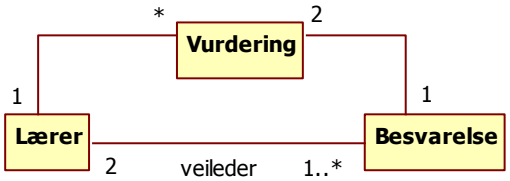


Pass på at kardinaliteten er minimum 0 i begge ender, ellers er blir det fort uendelig rekursivitet.

Assosiasjonsklasse

Hvis de to sensorene som er tilknyttet *Besvarelse* via assosiasjonen *sensurerer* er de samme to som har foretatt *Vurdering*, så er de to diagrammene like. Diagrammet til høyre er mer presist fordi det inkluderer denne forutsetningen.

I nedenstående diagram er det i virkeligheten *to* sammenhenger mellom *Besvarelse* og *Lærer* – den direkte *veileder* og den indirekte via klassen *Vurdering*. Lærerne som har vurdert er da ikke nødvendigvis de samme som har veiledet:

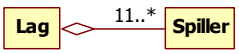


Da kan ikke *Vurdering* knyttes til assosiasjonen *veileder* som assosiasjonsklasse.

Merk at *objektering/entitetisering* slik man gjør under normalisering av datamodeller (ER/EAR) ikke er aktuelt i objektorientering, da mange-til-mange assosiasjoner er uproblematiske i objektorientering.

Aggregering (shared aggregation) og bestanddel (composition)

Aggregeringer og bestanddeler er bare forenklede notasjoner. Det dreier seg ellers om helt vanlige assosiasjoner.

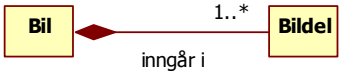


Betydningen av diagrammet ovenfor er at hver spiller er knyttet til flere lag (0..*) mens hvert lag består av flere spillere (her minst 11). Romben erstatter altså egentlig bare kardinaliteten 0..*. Man kan merke seg at spillere kan eksistere uten lag.

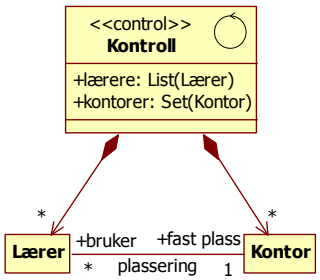
I diagrammet nedenfor hører hver bildel til i bare én bil, men det kan være flere (minst 1) bildel i hver bil. Den fylte romben erstatter kardinalitet 1..1, så en bildel kan ikke eksistere

[9] Refleksivitet har vi når noe refererer til seg selv. Rekursivitet har vi når noe gjentas ved å bruke seg selv. Assosiasjonen som er tegnet er altså rekursiv fordi den knytter en bildel en bildel. Den er rekursiv fordi dette kan gjentas så bildel 1 viser til bildel 2 som viser til bildel 3 osv. Skillet mellom rekursivitet og rekursjon gjøres ofte ikke særlig skarpt.

uten å inngå i en bil. Slettes bilen, må også alle dens bildeler slettes. Kardinaliteten for *Bildel* kan være en annen og assosiasjonene kan i prinsippet godt være enveis.



Jeg liker selv å tegne inn en kontrollklasse i mine diagrammer. I denne klassen legger jeg samlinger for de klassene som ellers inngår. Slik kan det se ut for et svært enkelt system:



(Da poenget her er assosiasjonene, er figuren forenklet. Jeg kommer senere nærmere tilbake til hvordan jeg gjør dette i detalj.)

Oppgave til kapittel 2 (relasjoner)

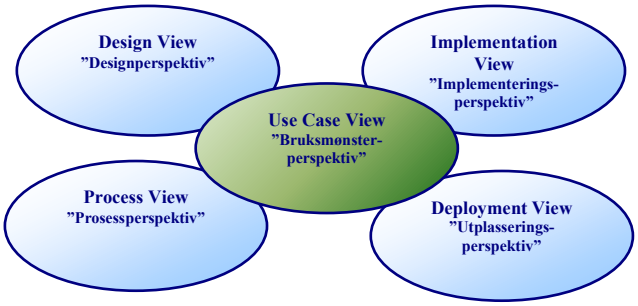
I denne oppgaven er vi bare opptatt av relasjoner i klassediagram. Det er ikke nødvendig å "finne på" attributter – ta bare med de som er nevnt i teksten. Det skal heller ikke spesifiseres operasjoner. Derimot skal alle assosiasjonene(en type relasjon) realiseres med passende assosiasjonsattributter.

Et verksted skal lage et OO datasystem for service som de gjennomfører. Det er gitt (eksempler i parentes):

1) Et *bilmerke* (navn = "Ford") er delt inn i *bilmodeller* (modellnavn = "Mondeo").
2) En bilmodell består av *deler* (delenummer = "abc3344"). For hver kombinasjon *bilmodell*⇔*del* lagres antallet av denne delen som inngår i bilmodellen i en *spec*. En del kan brukes i flere bilmodeller (en bestemt bolt brukes både i "Mondeo" og i "Fiesta"). Dette noteres altså for hver *bilmodell* og ikke for hver enkelt, individuelle bil (alle biler av denne bilmodellen anses å være like).
3) Delene settes ofte sammen til en overordnet del som altså da består av mange enkeltdeler ("bolt"+"mutter" = "komplett bolt", flere bolter = et "boltsett" osv.). En del kan inngå i flere slike sammensatte deler ("boltsett" inngår både i "motor" og i "sete").
4) Individuelle *biler* (regnr = "DA12345") er gjenstand for *service* (servicenr = "2009/17"). En service gjelder bare én bil, som selvsagt kan få service flere ganger.
5) Når det gjøres en service, deles den inn i *arbeidsoperasjoner*. Det registreres hvilke arbeidsoperasjoner som faktisk ble utført for denne servicen.
6) Til hver arbeidsoperasjon kan det være knyttet deler.
7) Objektene nevnt ovenfor, skal lagres i en database gjennom en kontrollklasse (merk denne klassen <<control>>).
8) Det skal lages to grensesnitt: En webbasert Applet og en GUI applikasjon.

Tegn klassediagram med alle relasjoner og spesifiser assosiasjonsattributtene som realiserer assosiasjonene.

Kapittel 3 – Use Case View: Bruksmønstre



Innledning

Som forklart i første kapittel, søker UML å beskrive systemet fra fem perspektiver/views. Sentralt er "Use Case View", vanligvis oversatt til "bruksmønsterperspektivet". Her analyseres systemet fra utsiden, det vil si at man er opptatt av hvilken hensikt interessentene har med systemet (jfr. det teleologiske prinsipp).

Begreper

Systemet under utvikling (**system under development** eller **SuD**) er det systemet man skal lage, og som man analyserer/designer. UML forutsetter at dette er et objektorientert system.

Interessenter (stakeholders) er alle som har en interesse av hvordan systemet "oppfører seg". Det kan være personer, avdelinger, andre systemer, organisasjoner osv., f.eks. skattekontoret som skal ha meldinger om skattetrekk og ledelsen som skal rapporter. De befinner seg i systemets omgivelser. Noen interessenter – men ikke alle – skal samhandle med systemet, og kalles da **aktører**. Det er aktørene som utgjør systemets omgivelser i systemteoretisk forstand. Interessent er altså en mer utvidet begrep, siden også noen utenfor omgivelsene kan ha interesse av systemets oppførsel.

Aktører (**actors**) er interessenter som samhandler med systemet. Aktørene har altså en oppførsel. Aktørene deles i primæraktører (primary actors) og støtteaktører (supporting actors). En aktør kan være primæraktør i forhold til et bruksmønster, og støtteaktør for et annet.

Primæraktører handler direkte med systemet og ofte – men ikke alltid – er det deres handlinger som tar initiativ i forhold til systemet og som systemet reagerer på. Systemet kan også reagere på andre hendelser, f.eks. at bestemte tidspunkt – slutten av måneden, bestemt klokkeslett o.l. – blir nådd eller at noen setter et kort inn i en kortleser. Disse bryterne (triggers) regnes ikke som aktører, siden de ikke har egen oppførsel. Noen primæraktører handler på vegne av andre, f.eks. en telefonselger som registrerer kjøp i systemet på vegne av en telefonkunde. Da kalles den som faktisk har interessen for den egentlige primæraktør (ultimate primary actor). Etter hvert som teknologien endres, vil gjerne den egentlige primæraktøren endres til primæraktør, som når kjøperen selv registrerer kjøpet på en webside.

Støtteaktører er aktører som utfører en eller annen tjeneste for systemet, f.eks. et fakturasystem som skal gi data til vårt system på forespørsel fra det.

Bruksmønstre (**use cases**) er en avsluttet samling handlinger som til sammen gjør noe av signifikant verdi (salient value) for en aktør. Det kan være produksjon av utdata, men like gjerne at systemets tilstand er endret, f.eks. at en ny karakter er registrert på en elev, at en elev er slettet osv.

Bruksmønstre representerer systemets **formål** (**goals**) og er dermed utgangspunkt for vurdering av systemet. Bruksmønstre har egenskaper. UML har ingen definitiv liste over slike egenskaper, men flere foreslår følgende:

- ✓ Navn: Navnet på bruksmønsteret
- ✓ Navnet på (del-)systemet som bruksmønsteret tilhører
- ✓ Interessent
- ✓ Primæraktør
- ✓ Kort beskrivelse
- ✓ Detaljeringsnivå: Grov oversikt (summary level), brukernivå (user level) eller subfunksjon nivå (sub function). Noen anbefaler ikoner – fra *sky*, via *sjø* til *musling* (figurene er gjengitt senere i kapitlet).
- ✓ Pre- og postbetingelser: Prebetingelser stiller krav om hva som må være sant *før* bruksmønsteret kan utføres – postbetingelser hva som bruksmønsteret skal garantere når det er ferdig utført. Postbetingelser kan være minimale garantier (alltid sanne) og suksessgarantier (sant hvis alt går bra).
- ✓ Brytere (triggers)

Scenarier (**scenarios**) er beskrivelser av handlingsmønstre. Det er en form for fortellinger, der aktørene og systemet inngår, og viser hva som skjer når aktøren bruker systemet for å oppnå sitt mål. De enkleste scenariene "ender godt" – alt går vel – og kalles hovedscenarier (**main success scenarios**). Man vet jo av erfaring at ting kan gå galt (Murphys lov) og derfor er det også behov for å håndtere unntakene (**exceptions**) og de beskrives som **utvidelser** (**extensions**).

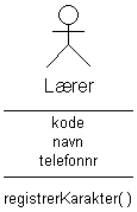
Hva beskrives i bruksmønsterperspektivet

Bruksmønsterperspektivet konsentrerer analysen om hvem/hva som utgjør systemets omgivelser og hvilken funksjonalitet systemet skal ha. Hensikten er å skaffe et overblikk over funksjonalitetskravene til systemet. Kravene gjelder kun hva systemet skal kunne gjøre, ikke hvordan. F.eks. spør man *ikke* etter sikkerhetskrav, kvalitetskrav, responstid, rammer i form av kostnader, tidsfrister, operativsystemer, maskinmiljø osv. Som kravspesifikasjon er det altså nokså snevert. Mange av de resterende kravene tar man hensyn til senere, når systemet utformes.

Omgivelser

Omgivelsene er mennesker og andre systemer som spiller en rolle i forhold til systemet man skal lage. De kalles derfor **actors** – vanligvis oversatt til **aktører** – men merk alluderingen til skuespillere som har en rolle. Aktører er altså ikke "Per" og "Kari", men "personalsekretær" og "personalsjef". Symbolet for en aktør er en "stickman":

Aktører benevnes med et **substantiv** som beskriver den **rollen de har i forhold til systemet**. Aktører er *klasser* (i objektorientert forstand) og kan følgelig *instansieres*, dvs. at det kan være mange av hver. F.eks. representerer aktøren *Lærer* mange lærere med de samme egenskapene. (Dette tilsvarer slik sett entitetstyper i datamodellering.) Siden aktørene er objektorienterte klasser, kan det knyttes både attributter (data) og metoder (handlinger som aktøren kan utføre på anmodning) til dem:



Aktørene vil senere bli representert inne i systemet ”rett innenfor systemgrensen” av klasser som håndterer den faktiske, fysiske brukeren (aktøren) på utsiden av systemgrensen. Slike klasser kalles **grenseklasser** (”boundary classes”). Gjennom et grensesnitt (MMI, GUI) gjør aktøren ønskene sine kjent: "Registrer karakteren 5 i engelsk på eleven Sylvester i klasse 2B”. Grenseklassen tar så – på vegne av aktøren – kontakt med andre klasser/objekter innover i systemet for å få dette utført.

Det er *ikke* vanlig å spesifisere aktørenes attributter og metoder før senere i analysen, og de vises uansett sjelden i bruksmønsterdiagrammene. De abstraheres bort (ref. prinsippet om ”need-to-know”).

Funksjonalitet

Systemets funksjonalitet defineres av hva aktørene vil at systemet skal gjøre for dem, hvilket tilsvarer systemets **formål**. Symbolet for et bruksmønster er en ellipse:

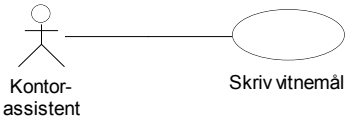
Siden bruksmønstre representerer en avsluttet handling, benevnes de med et verb, helst som en ordre i imperativ, som beskriver hva systemet skal gjøre for aktøren.

Relasjoner

Siden dette beskriver et system, er det relasjoner (sammenhenger) mellom delene. Det er flere, forskjellige relasjonstyper, nemlig assosiasjon, arv og avhengighet.

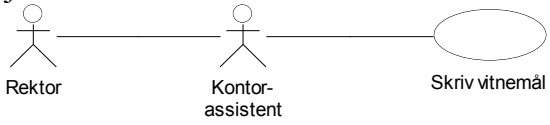
Assosiasjon

Relasjonen mellom en aktør og et bruksmønster, angir hvilken funksjonalitet aktøren ønsker = ”hvem ønsker hva”, og kalles *assosiasjon*. Symbolet for assosiasjon er en enkelt strek, uten retningsangivelse[10]:



Dette er den vanligste relasjonsformen i bruksmønsteranalysen. Legg merke til at dette ikke skal bety at kontorassistenten skriver vitnemål – dette dokumenterer at kontorassistenten vil at systemet skal skrive ut vitnemål når kontorassistenten ber om det.

Det kan også være assosiasjoner mellom to aktører:



[10] Enkelte forfattere har advokert for at assosiasjonen kan påføres en pil, for å angi hvem som *initierer* bruksmønsteret. Dette er *ikke* UML standard, og bryter med ”helhetsprinsippet” ettersom en pilsatt assosiasjon ellers innebærer en *enveis* sammenheng. Jeg kan selv ikke se behovet for å angi hvem som tar initiativ og er uansett imot en slik endring av pilens betydning.

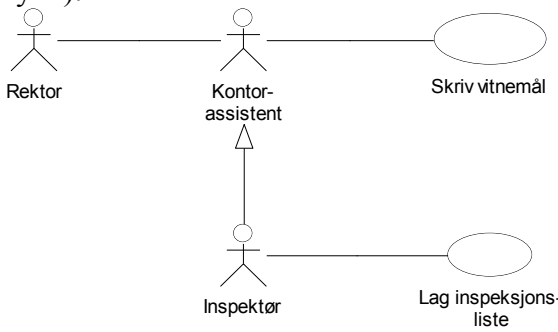
Dette er relativt sjelden og stort sett fordi brukere som er med på analysen insisterer. Det innebærer at det er rektor som er den egentlige interessent – kontorassistenten bare representerer rektor overfor systemet. Legg merke til at rektor ikke er i systemets omgivelser, og følgelig egentlig ikke skal være med i analysen. Kontorassistenten er primæraktør men rektor er den egentlige primæraktør slik det er definert ovenfor.

Det kan *ikke* være assosiasjoner mellom bruksmønstre.

Arv

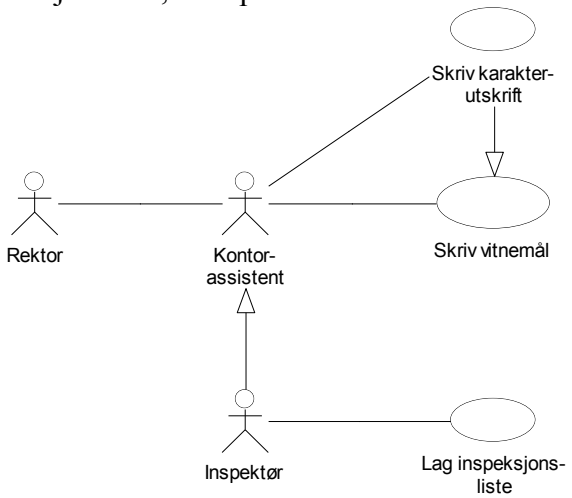
Arv innebærer at ett objekt overtar noen egenskaper fra et annet. Det objektet som arver, er subobjekt og det andre er metaobjekt (også kalt superobjekt).

Hvis en **aktør arver fra en annen aktør**, innebærer det at alle de ønskene metaaktøren har til systemet, har også subaktøren, men subaktøren har sine egne ønsker **i tillegg**. Symbolet for arv er en lukket pil (uten "fyll"):



Det kan være naturlig å tenke på arvehierarkier som et organisatorisk hierarki, men det bli helt feil. Dette er ikke et organisasjonskart, men viser at både kontorassistenten og inspektøren ønsker å få skrevet ut vitnemål, men i tillegg vil inspektøren få laget inspeksjonslister.

Hvis et bruksmønster arver fra et annet bruksmønster, innebærer det at subbruksmønsteret utfører omtrent samme funksjonalitet, men på en litt annen måte:

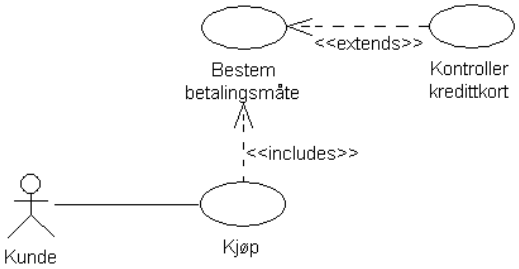


Å skrive en karakterutskrift er tydeligvis omtrent det samme som å skrive et vitnemål. Legg merke til at man plasserer symbolene fritt – det er ikke nødvendig at subobjektet plasseres nedenfor metaobjektet. Legg også merke til at symbolenes størrelse er uten betydning. Legg dessuten merke til at slik det nå er tegnet, er også inspektøren interessert i å få laget karakterutskrifter – det er arvet fra kontorassistenten.

Siden bruksmønstre og aktører er så forskjellige, er **arv mellom bruksmønstre og aktører** helt uaktuelt.

Avhengighet

Avhengighet oppstår bare mellom bruksmønstre og innebærer dypest sett at den avhengige kanskje må endres hvis den uavhengige blir endret. Mer spesifikt anvendes avhengighet for å uttrykke at et bruksmønster "bruker" et annet:



Et kjøp inkluderer alltid (="includes") å bestemme betalingsmåte, som på sin side kan ha behov for (="extends") å kontrollere kredittkort. Den prikkede, åpne pilen alene betyr avhengighet, men stereotypen angir en spesiell type (subtype) avhengighet.

Det er ikke ofte det er aktuelt med slike avhengigheter, og de kan lett overdrives. Eksempelet er ikke helt godt, fordi det er tvilsomt om kunden faktisk ønsker at systemet skal sjekke kredittkortet – er det virkelig en funksjonalitet som kunde vil at systemet skal kunne gjøre for ham/henne? Det kan også diskuteres om ikke "bestem betalingsmåte" bør sees som en naturlig del av et kjøp og inngå i den "avsluttede samling handlinger som til sammen gjør noe av signifikant verdi for en aktør" (jfr. ovenfor). Tegnemåten ovenfor kan likevel være aktuell hvis andre aktører spesielt har ønsket seg *kontroll av kredittkort* (alene).

Sammenheng mellom bruksmønsterdiagrammer

Bruksmønsterdiagrammer blir fort "overlesset" med for mange objekter og relasjoner, så man mister oversikten. Siden et uttrykkelig formål med bruksmønsteranalysen er å skaffe overblikk over aktørenes ønsker for systemet, må diagrammene da deles. En vanlig regel er jo "maks 7±2" objekter pr graf – kanskje ni ved enkle relasjoner, men ned mot fem ved komplekse.

Det er ingen notasjon for å angi sammenheng mellom diagrammene, som alle er på samme nivå. Det ene bruksmønsterdiagrammet er altså ikke et subdiagram til et annet (meta)diagram. Man kan velge hvordan man vil inndele diagrammene:

- ✓ Tegne en aktør bare på ett diagram. Alle bruksmønstre denne aktøren er interessert i tegnes inn der (hvis det er mulig – det kan jo være mange). Bruksmønstre må da ofte gjentas på flere diagrammer.
- ✓ Tegne et bruksmønster bare på ett diagram. Alle aktørene som er interessert i dette bruksmønsteret tegnes da inn der (hvis det er mulig). Aktører må da ofte gjentas på flere diagrammer.
- ✓ Forsøke å dele systemet opp i "naturlige deler" og tegne alle aktører og bruksmønstre som tilhører en gitt del inn på samme diagrammer.

Det er sikkert andre strategier som jeg ikke har tenkt på. Poenget er at man tilslutt får overblikk.

Detaljering med tekst

Det er nødvendig å spesifisere aktører, bruksmønstre og relasjoner ytterligere. Hensikten er da ikke å ta beslutninger om realisering, men å gi ytterligere forståelse. Det vanligste er å gjøre dette ved å knytte en vanlig tekst til forholdet, f.eks. ”Det er rektor som skal ha vitnemålene, for kontroll og underskrift.” eller ”Kontroll av kredittkort gjelder bare nye kunder, men gjøres også årlig for alle faste kunder”.

Til beskrivelse av bruksmønstre, brukes gjerne *maler*. Her er et eksempel på en slik mal, hentet fra Rational Unified Process (RUP):

Use Case xxx:

Overview:
Notes:
Actors:
Preconditions:
Scenario:

No.	Action (Stimulus)	Software Reaction
1		
2		
3		

Scenario Notes:
Post Conditions:
Exceptions:
Required GUI and GUI Sketches:
Dependencies and Relations:

Her er et eksempel på et scenario, skrevet med syntaks etter Cockburn[11]:

Prebetingelse
Automaten viser teksten ”Sett inn kort”

Hovedscenario (suksess)
1) Sjåføren drar kortet sitt i parkeringsautomaten
2) Automaten sjekker at kortet er gyldig
3) Sjåføren angir parkeringstid
4) Automaten viser utløpstid
5) Sjåføren trykker knappen ”Billett”
6) Automaten usteder billett
7) Bruksmønsteret avsluttes med suksess

Utvidelser (unntak)
2a) Kortet er ugyldig
- 2a1) Automaten viser teksten ”Ugyldig kort” i 10 sekunder
- 2a2) Automaten viser teksten ”Sett inn kort”
- 2a3) Bruksmønsteret avsluttes

2b) Kortet er uleselig
- 2b1) Automaten viser teksten ”Uleselig kort” i 10 sekunder
- 2b2) Automaten viser teksten ”Sett inn kort”
- 2b2) Bruksmønsteret avsluttes

5a Sjåføren trykker knappen ”Avbryt
- 5a1) Automaten viser teksten ”Avbrutt”
- 5a2) Bruksmønsteret avsluttes

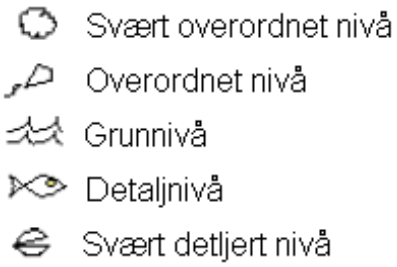
Postbetingelse
Minimum: Automaten viser teksten ”Sett inn kort”
Suksess: Automaten viser teksten ”Sett inn kort” og har utstedt billett.

Bruksmønstre kan også detaljeres med aktivitetsdiagrammer (”activity diagrams”). De likner på (men er ikke helt som) de gamle flytdiagrammer (”flow charts”). Jeg går ikke nærmere inn på dem her men viser eksempler senere (kapittel 6) i en annen sammenheng.

Pakking
Det er mulig å ”pakke” inn ett eller flere bruksmønstre i en ”pakke”. Dette gjøres kun for å lette oversikten, og gir fortsatt ingen prinsipiell nivåinndeling. Jeg velger å ikke ta med dette her.

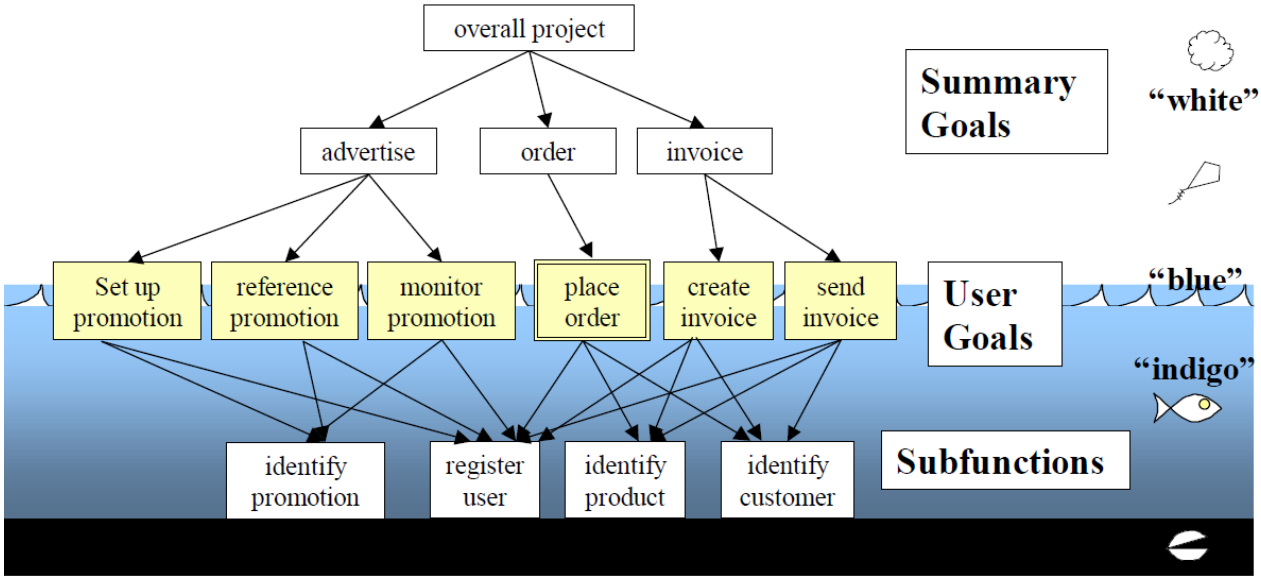
Oversikter over bruksmønsterdiagrammene
Når det er sammenheng mellom bruksmønsterdiagrammer, kan det være aktuelt med en egen graf som viser sammenhengene. Her er et eksempel på syntaks hentet fra Oracle:

[11] Alistair Cockburn: ”Writing Effective Use Cases”, Addison-Wesley, 2001, ISBN 0-201-70225-8



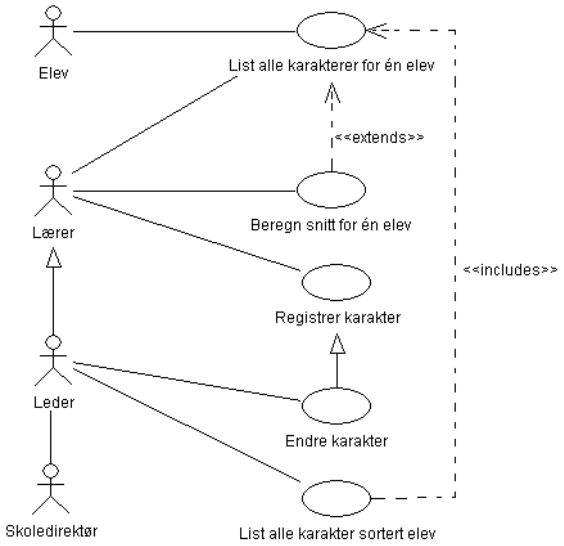
Kilde: Oracle White Paper: "Getting Started With Use Case Modeling", 2007

Grunnivået er brukernes nivå, det er her de får beskrevet hvilken funksjonalitet de ønsker at systemet skal ha[12]. Cockburn mener at det nederste nivået er *for* detaljert. Her er et eksempel:



Kilde: Cockburn, op.cit.

Et eksempel på bruksmønsterdiagram

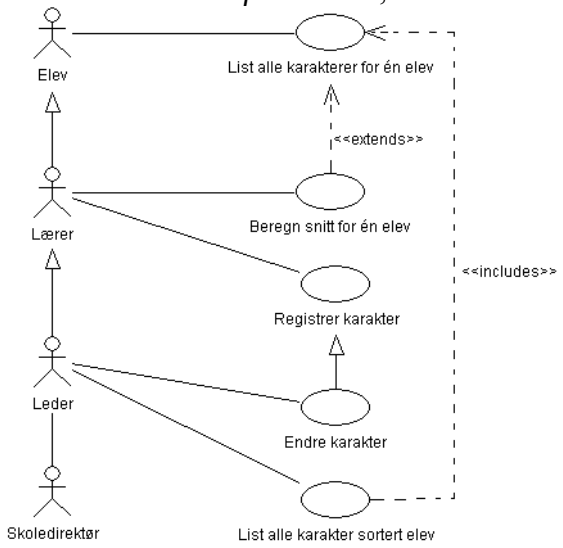


Note: I eksemplet ovenfor er det tatt med alle mulige sammenhenger for vise all bruk. Det vanlige er at bruksmønsterdiagrammer bare består av aktører, bruksmønstre og assosiasjoner mellom aktørene og bruksmønstrene. Andre relasjoner er uvanlige og bør benyttes med stor

[12] Det kan diskuteres om "Log On" er et godt bruksmønster – se neste side.

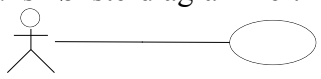
forsiktighet. Alistair Cockburn mener man ikke bør bruke ”extends”, bare ”include” som han sammenlikner med et vanlig subrutinekall.

Assosiasjonen *Lærer⇔List alle karakterer for én elev*, kan erstattes av arv *Lærer⇨Elev*:



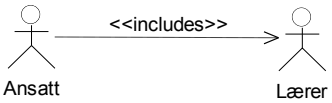
Noen typiske feil

Nybegynnere (f.eks. studenter) gjør enkelte typiske feil når de tegner bruksmønsterdiagrammer. Her er noen eksempler:

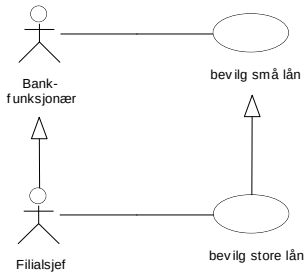


Hvem er denne aktøren? Hva vil han at systemet skal gjøre, egentlig?

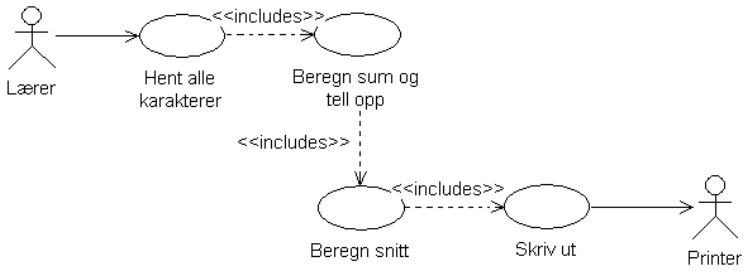
Er *Logg inn* virkelig en avsluttet samling handlinger som til sammen gjør noe av signifikant verdi for en aktør”? (Men man skal ikke se bort fra at ”Kontroller om brukeren har brukstillatelse” *kan* være en ønsket funksjonalitet.)



Dette vil innebære at *Lærer* skal settes inn i *Ansatt* på et passende sted. Interessant – men neppe det som er ment. Hva *er* egentlig ment?



Meningen er nok at bankfunksjonær kan bevilge små lån, men bare filialsjef kan bevilge store lån. Da er dette feil!
Siden ”bevilg store lån” er en spesialvariant av ”bevilg små lån”, kan bankfunksjonær utføre *begge*.



Den siste figuren er *helt* galt, men varianter av denne sees ofte fra studenthold. Tegneren tenker seg nok en *sekvens av handlinger* (først hentes alle karakterene, så beregnes sum og antallet telles opp, så…). *Printer* er ikke en aktør – den ønsker ikke funksjonalitet fra systemet. Muligens kan den hevdes å være støtteaktør, men jeg vet ikke hvilke krav til funksjonalitet en printer som stille til systemet – det er vel heller omvendt. Enveis assosiasjoner (f.eks. *Lærer⇨Hent alle karakterer*) er ikke tillatt i bruksmønsterdiagrammer. Dessuten innebærer «includes» at det ene bruksmønsteret *inkluderer* det andre, ikke at det overfører kontrollen til det. Ifølge dette diagrammet vil altså f.eks. *Beregn snitt* inkludere *Skriv ut* og det er neppe det som er ment.

Antagelig mener systemereren her helt enkelt at *Lærer* ønsker bruksmønsteret *Skriv karakterstatistikk* eller liknende. *Hvordan* det skal gjøres er temmelig trivielt, men kan eventuelt spesifiseres med tekst eller aktivitetsdiagram, og ikke i bruksmønsterdiagrammet.

Sammenhengen mellom bruksmønster- og det logiske perspektivet

Bruksmønsterperspektivet er svært sentralt i UML. Det vil føre for langt her å gå inn på alle sammenhenger mellom perspektivene. Sammenhengen til det logiske perspektivet er imidlertid antakelig det viktigste, og om denne sammenhengen vil jeg derfor trekke frem noen få momenter knyttet til overgangen mellom bruksmønsterperspektivet og det logiske perspektivet.

Det logiske perspektivet dokumenterer systemet innenfor systemgrensen, og det må naturligvis passe med dets utside som er dokumentert med bruksmønsterdiagrammer. Man vil forvente at systemets deler – i samarbeid – kan utføre alle bruksmønstrene.

Videre vil man forvente at bruksmønsterperspektivets aktører, er representert ved klasse/objekter innenfor systemet (som grenseklasser, ”boundary classes”), gjerne med andre navn.

Bruksmønstrene representerer *funksjonalitet*, men ofte vil funksjonalitet implisitt kreve datalagring. F.eks. vil bruksmønsteret *”Lage karakterliste”* kreve at systemet lagrer karakterer og andre opplysninger som skal være med på listen. Man forventer at det logiske perspektivet tar høyde for det.

Drøfting

Hva beskrives *ikke* i bruksmønsterperspektivet

Et viktig aspekt når man ser et system ”fra utsiden”, er hvordan samarbeidet foregår mellom aktørene og systemet, altså **grensesnittet** mellom dem.

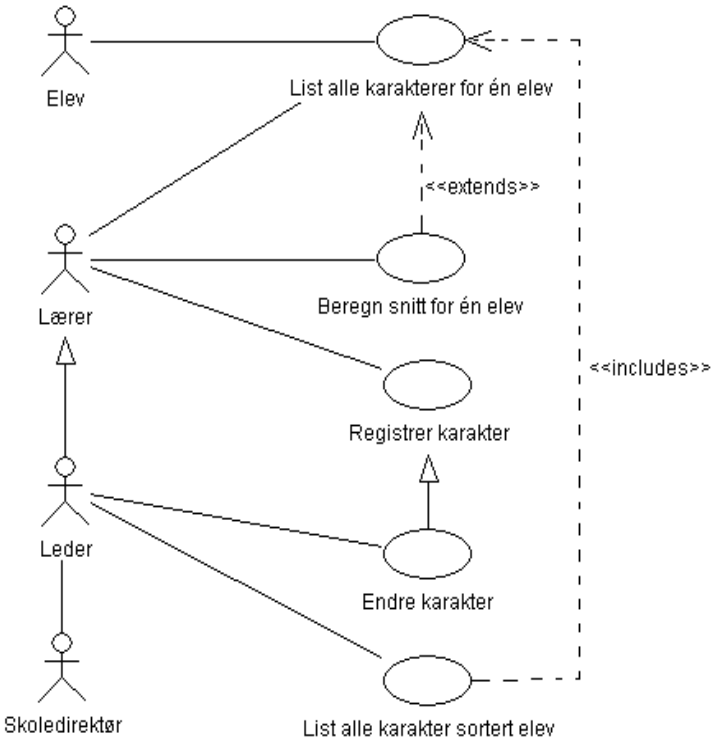
Hvis aktøren er et annet *system*, vil det dreie seg om en (temmelig teknisk) protokoll (= en mengde formaliserte meldinger og signaler). Den vanlige dokumentasjonsformen for slikt, er tekster og tilstandsdiagrammer.

Hvis derimot aktøren er et *menneske* (en *bruker*) vil det være behov for å spesifisere MMI (”Man Machine Interface” = Menneske maskin interaksjon) evt. GUI (”Graphical User Interface” = grafisk brukergrensesnitt). UML har ingen dokumentasjonsform for dette, men anbefaler at man lager prototyper. Sannsynligvis vil det være fornuftig tidlig i utviklingen å lage prototyper av typen ”mock ups”, altså prototyper bare med grensesnitt og uten funksjonalitet.

Fordeler og ulemper med bruksmønsterdiagrammer

Bruksmønsterdiagrammer kan sees som en funksjonsbeskrivelse, som utgjør en del av kravspesifikasjonen. Slike funksjonsbeskrivelser kan man også enkelt lage som lister:

Graf (diagram)



Liste

Elev ønsker:
1) List alle karakterer for én elev – benytter om nødvendig
   a) Beregn snitt for én elev

Lærer ønsker:
1) Det samme som **Elev**
2) Beregn snitt for én elev
3) Registrer karakter

Leder ønsker:
1) Det samme som **Lærer** med tillegg av:
2) Endre karakter – variant av Registrer karakter
3) List alle karakterer sortert elev – benytter alltid
   a) List alle karakterer for én elev – benytter om nødvendig
      i) Beregn snitt for én elev

Skoledirektør representeres av **Leder**.

Det har imidlertid en del fordeler å benytte grafer:
1) Det er mer visuelt. Mange oppfatter og jobber bedre med bilder enn med tekst.
2) Grafer inviterer til kreativitet. Det er viktig å få alle med på det skapende arbeidet som kreves i analysearbeid.
3) Grafer kan gi bedre overblikk. Særlig kommer sammenhenger ofte bedre frem.
4) Grafer tvinger frem en bestemt måte å tenke på. Språket i en graf har svært få symboler (”ord”) og det som kan uttrykkes er begrenset. Dette bidrar til å konsentrere arbeidet om noen få aspekter ved systemet. Dette kan være begrensende, men vanligvis er systemene så komplekse at det er god strategi å dele opp problemet og ”se på en ting av gangen”.
5) Grafer er presise. Det er meget enklere å være uklar, vag og usikker i en tekst enn i en graf. Dette tvinger frem en diskusjon om ”hvordan det egentlig er”.

Men grafer har også ulemper.

1) Grafene har sin egen syntaks som må kunnes.
2) Syntaksen kan komme i veien for arbeidet. Man vet hva man vil si, men finner det vanskelig å få uttrykt det.
3) Grafer krever i praksis et tegneprogram – helst ett som har syntaksen programmert inn. Grafingen krever da også at man behersker tegneprogrammet, og at programmet har tilstrekkelig kvalitet. Det er vanligvis enklere – og billigere – å benytte tekstbehandling, og man unngår versjonsproblematikk.

Det er ikke uten videre gitt at utviklingsarbeidet bør begynne med bruksmønsterperspektivet. (Andre utviklingsmetoder har begynt med dataanalyse, med virksomhetsanalyse, rutineanalyse, organisasjonsanalyse og annet.)

Innledende analyse – et enkelt hjelpemiddel

Som hjelpemiddel for å komme litt innpå et nytt system, kan det være en idé å lage en enkel oversikt, som jeg har laget og brukt med suksess. Jeg kaller det ABE-tabell av åpenbare grunner.

Aktører	**Bruksmønstre**	**Entitetstyper**
Selger Kjøper	Legg inn produkt Bestem tilslag Legg inn bud	Produkt Produktkategori? Selger? Kjøper? Felles person?? Bud

Figuren viser en svært tidlig fase av "idédugnad" for et auksjonssystem.

I en idédugnad fyller man inn ABE-tabellen uten tanke på om den er riktig (i henhold til reglene for idédugnader). Først etterpå rydder man opp ved å sikre at aktørene, bruksmønstrene og entitetstypene er i riktig kolonne. Videre at de faktisk er det de gir seg ut for. F.eks. kan ofte flere foreslåtte entitetstyper representere det samme (vare, produkt), flere bruksmønstre skal egentlig gjøre det samme (skriv regning, skriv faktura) og aktører har ikke egentlig kontakt med systemet, eller aktørene er de samme (kunde, kjøper). Dessuten kan ofte forslagene slås sammen ved at et forslag inngår i et annet. ABE-tabellen gir erfaringsmessig godt utgangspunkt for slike diskusjoner.

Oppgave til kapittel 3 (SOS-Forum)

Det skal lages et edb-basert "SOS-forum" (**S**pørsmål-**O**g-**S**var). Bare "medlemmer" av forumet kan delta, men alle (public) kan lese innleggene.

Forumet har en administrator, vanligvis læreren. Administrator har visse spesielle rettigheter – se nedenfor – men er ellers å regne som et vanlig medlem.

For å bli medlem, må man be om det i en egen del av forumet. Administratoren vurderer alle søknader om medlemskap en gang i blant, og avviser eller godkjenner hver enkelt. Først når medlemskapet er godkjent, kan medlemmet begynne å legge inn spørsmål og/eller svar.

Forumet virker slik at et medlem stiller et spørsmål. Spørsmålet sies da å være ”åpent”. De som mener å vite svaret legger det inn.. Når spørreren eller administrator anser spørsmålet som ”besvart”, noteres det av en av de to, og deretter er det ikke lenger tillatt å legge inn flere svar på dette spørsmålet.

Medlemmet må oppgi om han/hun ønsker å legge inn et spørsmål, eller om det er et svar og da til hvilket spørsmål (kun ett).

Strukturen er altså at det ligger en del spørsmål lagret i forumet. Noen av dem er ”åpne” og kan besvares. Det kan gis mange svar til hvert spørsmål. Åpne spørsmål synes i åpningsbildet, sammen med svarene som er gitt. Spørsmål som er ”besvart” synes ikke automatisk, men de kan vises ved å søke. Besvarte spørsmål kan ha ett (minimum) eller flere svar.

Administrator kan fjerne spørsmål og alle dets svar, f.eks. hvis spørsmålet anses upassende eller utenfor forumets tema. Administrator kan også slette medlemmer, eller sperre dem for en viss tid (fra straks og inntil en viss dato).

Eksempel på utlisting av spørsmål med søkeordet ”gammel”:

Det er funnet 3 spørsmål vedr. ”gammel”:

Besvart Spørsmål 32 (2 svar):
Hvor **gammelt** er UML?
Erik
Svar 32-1:
Knut sa noe om slutten av 1990-årene.
Kari
Svar 32-2:
Følgende finnes på http://www.omg.org/gettingstarted/what_is_uml.htm:
”Identifiable object-oriented modeling languages began to appear between mid-1970 and the late 1980s as various methodologists experimented with different approaches to object-oriented analysis and design. The number of identified modeling languages increased from less than 10 to more than 50 during the period between 1989-1994. Many users of OO methods had trouble finding complete satisfaction in any one modeling language, fueling the "method wars." By the mid-1990s, new iterations of these methods began to appear and these methods began to incorporate each other’s techniques, and a few clearly prominent methods emerged.

The development of UML began in late 1994 when Grady Booch and Jim Rumbaugh of Rational Software Corporation began their work on unifying the Booch and OMT (Object Modeling Technique) methods. In the Fall of 1995, Ivar Jacobson and his Objectory company joined Rational and this unification effort, merging in the OOSE (Object-Oriented Software Engineering) method. ……

The efforts of Booch, Rumbaugh, and Jacobson resulted in the release of the UML 0.9 and 0.91 documents in June and October of 1996. ……

This collaboration produced UML 1.0, a modeling language that was well defined, expressive, powerful, and generally applicable. This was submitted to the OMG in January 1997 as an initial RFP response. ……

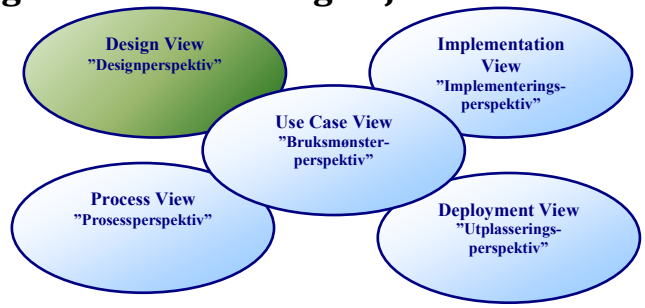
The focus of the UML 1.1 release was to improve the clarity of the UML 1.0 semantics and to incorporate contributions from the new partners. It was submitted to the OMG for their consideration and adopted in the fall of 1997."
Petra

Åpent spørsmål 43 (1 svar):
Hvor **gammel** er Knut?
Olga
Svar 43-1:
Jeg er da ikke gammel!
Knut

Åpent spørsmål 61 (0 svar):
Er denne skolen **gammel**?
Petter

Vi lager en analyse sammen i timen – dere tegner bruksmønsterdiagrammer med StarUML til neste uke.

Kapittel 4 – Design View: Klasser og objekter



Objekter vs klasser

Objekter…

1. *Tar vare på data*
   Dataene kaller UML for *objektattributter* eller oftest bare *attributter*.
2. *Kan gjøre ting (har en oppførsel)*
   Handlingene kaller UML for *objektoperasjoner* eller som oftest bare *operasjoner*.
3. *Har en tilstand*
   Tilstanden er verdien av alle data sett under ett + evt. programteller hvis objektet eksekverer en operasjon
4. *Har identitet*
   Identiteten er gitt ved en adresse (vanligvis til lokal RAM). To objekter er like pr definisjon, hvis de har samme adresse i RAM, det vil i praksis si at det er samme objekt.
5. *Kan ta imot og sende meldinger*
   Å "sende melding" er det samme som å kalle et objekts/klasses medlem. Melding må ha en adresse (objektet/klassen som meldingen sendes til) og navnet på et synlig medlem evt. med korrekte parametre. Adressen er objektets/klassens identitet. For å sikre tilgang til et objekt, må man ha en referanse til det – i Java gjøres det med variable som sette til å peke til objektet med *new*.
   ```
   Student nyStud = new Student();
   ```
6. *Kjenner til klassen de tilhører*
   Man vet jo at objektene må ha egen plass i RAM til dataene, for de er forskjellige for hvert objekt. Det er imidlertid svært redundant å lagre de kompilerte operasjonene i hvert objekt. De lagres isteden bare i klassen. Derfor må et objekt også kjenne til sin klasse.
7. *Har en varighet (levetid)*. De kan være
   a. *Transiente*
      Objektene skapes ved behov og slettes senest når programmet avslutter
   b. *Persistente*
      Objektene skapes ved behov og overlever mellom sesjoner gjennom en form for lagring på ytre lagringsmedium (fil, database)

Klasser…

1. *Virker som mal*
   Når et objekt skal skapes – det heter at klassen *instansieres* – brukes klassen til å bestemme hvilke attributter objektet skal ha og følgelig hvor mye plass som skal settes av i RAM og hvordan.
2. *Har konstruktør og destruktør*
   Konstruktøren brukes når klassen skal instansieres. I Java heter de henholdsvis klassenavnet og *finalize*. I VB kjenner man dem som New og Finalize. I C++ heter de

det samme som klassenavnet og ~klassenavnet.
Begge blir automatisk lagt til av kompilatoren (ikke i koden men i den kompilerte koden) hvis man ikke har laget egne. Hvis man har laget våre egne, vil de overstyre automatikken. De er vanligvis synlige public, men noen språk tillater også private. Minst én konstruktør må allikevel vanligvis være public, ellers er det ikke mulig å skape objekter "utenfra"[13].
I noen språk (Java, VB, C#) kalles destruktøren av en garbage collector i egen tråd når objektet ikke lenger har noe som refererer til det, med lite kontroll på når det skjer. Det advares av mange på det sterkeste å kalle *finalize* selv i Java. Egne definisjoner av *finalize* aksepteres som nødvendig i noen tilfeller, men det vil forsinke eksekveringen.
I C++ (men ikke C#) kan destruktøren kalles uttrykkelig med verbet *delete* og det gjøres rutinemessig.

3. *Tar vare på egne data*
   UML kaller dem *klasseattributter*
4. *Kan gjøre ting (har en oppførsel)*
   UML kaller dem *klasseoperasjoner.*
5. *Tar vare på kompilert kode for alle operasjonene*
   a. Objektoperasjoner som bare objektene kan utføre
   b. Klasseoperasjoner som bare klassen selv kan utføre
6. *Har identitet*
   Klassens identitet er dens navn og skrives vanligvis med stor forbokstav. Det utgjør samtidig en referanse til klassen i RAM. Klassene lastes inn som en del av programmet og blir der sålenge programmet eksekverer.
7. *Kan ta imot og sende meldinger*

```
    int i = Integer.parseint(tekst); //klasseopersjonen parseint i
Integer
    i = Integer.MAX_VALUE; //MAX_VALUE er et klasseattributt i
Integer
```

8. *Kan arve attributter og operasjoner*
   Klassen kan arve fra metaklasser (i Java/VB/C++ gjør de det alltid) og de kan arves til subklasser. Java er strengt hierarkisk, C++ kan ha multippel arv. Det eneste som ikke arves er konstruktøren/destruktøren – det krever noe annet å skape et objekt av en subklasse.
   Man kan
   a. Overstyre (override) en arvet operasjon, dvs. deklarere operasjonen på nytt med samme signatur og gi den ny virkemåte. Signaturen inkluderer navn etterfulgt av type og rekkefølge av parametre, men ikke operasjonens type, synlighet eller exceptions. Dette gir *polymorfisme* ved at meta- og subklassen oppfører seg forskjellig etter samme melding.
   b. Definere arvede, abstrakte operasjoner (skrive kode for dem) eller la dem forbli abstrakte og deklarere også subklassen abstrakt.
   c. Legge til nye medlemmer.
   d. Legge til nye operasjoner med samme navn men med ny signatur, dvs. overlaste (overload).
   e. Endre synlighet av arvede medlemmer, men bare utvide den, f.eks. fra private til public. (Det er fordi kompilatoren må sjekke synlighet for metaklassen når et objekt brukes, selvom objektet faktisk viser seg under run-time å være av en subklasse.)

[13] Når man lager "singletons", dvs klasser som kun skal instansieres til ett objekt, lager man bare én, privat konstruktør – se eget avsnitt om singletons senere i kapitlet.

f. Overstyre attributter. Dette kalles shadowing og er ikke å anbefale, da objektene i så fall får flere attributter med samme navn.
g. *Ikke* fjerne arvede medlemmer.

9. *Klasser deles i forskjellige typer etter hva de modellerer*
   Typen er ikke særlig viktig, men den hjelper til å finne gode klasser og gir signal om hvilke som bør være persistente. De tre vanligste typene er definert med stereotyper (se også slutten av dette kapitlet):
   a. *Entity class* modellerer interessante ting i omgivelsene
   b. *Boundary class* håndterer omgivelsene og er deres "representant" innenfor systemgrensen, f.eks. brukere, andre systemer, databaser, printere
   c. *Control class* styrer samarbeidet mellom klasser/objekter. Vanligvis er de transiente og gjerne knyttet til et bruksmønster (så kontrollklassen sørger for å få gjort en arbeidsoperasjon som omgivelsene er interessert i å få gjort).

   Andre typer finnes også, som f.eks. *interface*, *auxiliary*, *signal* og *enumeration*.
10. *Kan være abstrakte eller konkrete*
    a. Konkrete klasser kan instansieres. De må følgelig ha minst ett objektattributt eller én objektoperasjon og ingen abstrakte objektoperasjoner.
    b. Abstrakte klasser kan ikke instansieres, men er gjerne gjenstand for arv. Klasser med minst én abstrakt operasjon, *må* være abstrakte.

Attributter og operasjoner

Attributter og operasjoner kalles med et fellesord for *medlemmer*. Helt presist bør man skille mellom *klassemedlemmer* (klasseattributter og klasseoperasjoner) og *objektmedlemmer* (objektattributter og objektoperasjoner). Ordet *medlemmer* (attributter og operasjoner) bør da bare brukes som et fellesord for objekt- og klassemedlemmer. I praksis er man sjelden så presis.

Attributter: Dataene som klasser og objekter tar vare på kalles attributter. Attributter er alltid konkrete, men de kan være objektattributter eller klasseattributter. De må ha unike navn innen klassen og skrives vanligvis med liten forbokstav. Det er vanlig å skille ordene i et sammensatt navn enten med understrek (*lag_noe*) eller med stor forbokstav "camelCase" (*lagNoe*).

Attributter kan være variable eller konstanter (*final* i Java). I noen språk (inkludert Java[14]) er det vanlig å skrive konstanter med bare store bokstaver og understrek (*Integer.MAX_VALUE*).

Det regnes som god skikk å deklarere alle attributter *private*. Det gjennomfører både informasjonsskjuling og innkapsling. En stor fordel ved å gjøre det slik, er at all tilgang til attributtet "utenfra" må gå igjennom en synlig operasjon. Der legger man kontroll, konvertering o.a. som da blir skrevet bare ett sted så man unngår redundans.

Operasjoner: Operasjoner kan være klasseoperasjoner eller objektoperasjoner. De første kan kun kalles med en melding til klassen, de siste kan bare kalles hos objekter. Bare objektoperasjoner kan være abstrakte. De kan bare finnes i abstrakte klasser.

Når attributtene er deklarert som private, men likevel skal være tilgjengelige "utenfra", må det lages operasjoner som gir tilgang til attributtverdiene. For et enkelt attributt (f.eks. int, boolean, String) trengs det minst én metode for å hente verdien, ofte navngitt med prefikset

[14] Interesserte finner god programmeringsskikk for Java beskrevet av Sun på http://www.oracle.com/technetwork/java/codeconv-138413.html

get (*hent*), og én metode for å gi noe ny verdi, navngitt med prefikset *set* (*sett*) og den nye verdien som argument. Slike operasjoner kalles *tilgangsoperasjoner* fordi de gir tilgang til skjulte data. Noen attributter bør bare kunne leses og ikke endres fra utsiden ("read only") eller være umulig å endre (*immutable*). Det kan f.eks. ordnes ved å ha bare get-operasjon eller privat set-operasjon. Alle avledede attributter (som bare beregnes eller finnes på forespørsel) er av denne typen – de kan jo ikke endres ettersom de ikke lagres.

Et spørsmål er om også klassens/objektets egne operasjoner skal bruke tilgangsmetodene. Jeg mener det er det sikreste – også når tilgangsmetoden hverken konverterer eller kontrollerer. Det forenkler vedlikehold. I noen få tilfeller er det riktignok ikke tillatt, så da får man heller dublere kode.

Tilgangsoperasjonene kalles populært for "getter- og setter-metoder".

Klassediagrammer

Ute i virkeligheten omkring oss har man objekter, dvs. "ting". Det kan være f.eks.

- ✓ Konkrete ting (bok, eksamensbesvarelse, bil)
- ✓ Abstrakte ting (fag, kurs, ekteskap)
- ✓ Roller (student, lærer, ektefelle, eier)
- ✓ Hendelser (bryllup, eksamen, ankomst)
- ✓ Interaksjoner (kjøp, møte)

Det spiller ingen rolle for analysen hva slags "ting" det er snakk om.

En viktig type av klasse/objekt er de som modellerer disse "tingene" ute i virkeligheten. Modellen lages elektronisk – som et objekt – og skal representere hvert sitt eksemplar av de "ting" systemet vårt interesserer seg for. Man modellerer selvsagt bare "ting" man er interessert i, og samler bare opplysninger som interesserer oss (i Universe of Discourse, UiD). Operasjonene kan modellere den virkelige "tingens" oppførsel ("giftDeg(ektefelle)") men mange av operasjonene er lagt til for å få systemet til å virke som det skal.

F.eks. har vår venn Petter mange egenskaper: navn, vekt, alder, fødselsdato, omkrets rundt overarmen, blid, grei, flink i fotball, sønn av Nils og Kari, samboer med Olga osv. Man er kanskje bare interessert i et nr (som man lager selv) og navnet.

Alle objektene under ett utgjør vår modell og er vår "database", men – OBS! – den har oppførsel og den ligger i RAM.

Hensikten med modellen er å avspeile virkeligheten, slik at endringer/hendelser i virkeligheten som gjelder Petter, gir tilsvarende endringer/hendelser inne i vårt system. Da kan man finne ut ting om virkeligheten ved å sjekke i modellen. Isteden for å finne Petter og spørre ham om hvilket nr han har, kan man finne ham (elektronisk) i modellen og "spørre" ham der ved å sende en melding til objektet som representerer Petter.

I klassediagrammer tegner man først og fremst klasser og relasjoner (se eget notat om relasjoner) mellom dem. I tillegg kan det (sjelden) tegnes inn grensesnitt (som en klasse med stereotypen «interface» eller som liten sirkel), objekter og pakker. Det brukes ofte kommentarer og noter som tilknyttes ett element. UML har egne symboler for mange av disse.

De vanligste relasjonene er mellom klasser og inkluderer *assosiasjoner* (inkludert *assosiasjonsklasser*), *arv* og *avhengigheter*. Det finnes flere, men de brukes mest i tekniske systemer og tas ikke med her.

Systemgrensen tegnes ikke inn klassediagrammer – den forutsettes å gå i ytterkanten av modellen da den bare dokumenterer systemets *innside*. Systemet sett fra utsiden er dokumentert i bruksmønsteranalysen.

Komponenter

Et hovedargument for å bruke objektorientering, er at man kan gjenbruke klasser i andre sammenhenger. Klassene skal da helst være "pluggable", slik at man bare kan legge inn en klasse man har og dermed er det problemet løst. I praksis er det meget vanskelig unntatt for lavnivå klasser som f.eks. Java-bibliotek. Ellers varierer design, teknologi, omgivelser, arkitektur osv. for meget til at det går. Det er lite sannsynlig at f.eks. en klasse Student laget til eksamenssystemet vil passe i regnskapssystemet, men noen ganger kan det ordnes med arv.

Tilleggsinformasjon i diagrammene

Elementer kan merkes med forskjellig informasjon i tillegg til navnet. Stereotyper, noter, restriksjoner og kommentarer er omtalt tidligere (kapittel 2). Her tas derfor først og fremst annen informasjon.

Synlighet

Elementer kan merkes med synlighet. Predefinert i UML er følgende fire:

- \+ public
- \- private
- \# protected
- ~ package

I tillegg har jeg sett følgende brukt:

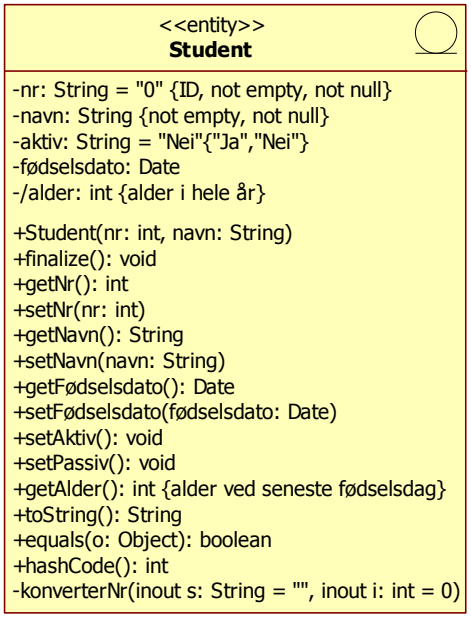
- / avledet (et attributt som ikke lagres)

Det som er synlig utenfra, utgjør et objekts grensesnitt. Synligheten varierer etter hvor man ser fra. Derfor bør man egentlig snakke om "public interface", "package interface" osv.

For å opprettholde innkapsling, gjør man synligheten minst mulig.

Eksempel

Her er et eksempel på en student (dokumentert med StarUML):



Om klassen

Klassen tegnes som et rektangel. Det er delt i tre "compartments", avdelinger(?) og det kan legges til flere, f.eks. felt for ansvar, regler og endringshistorie. De tre avdelingene navn, attributter og operasjoner er standard – legger man til flere bør de navngis.

I navnefeltet fremgår det med fet skrift at klassen heter "Student". Av stereotypen <<entity>> og ikonet, ser man at klassen er en entitetsklasse.

Klassen er konkret. Abstrakte klasser må merkes spesielt med stereotyp og/eller navnet skrevet i kursiv.

Om attributtene

I attributtfeltet ser man at objektene har fem attributter: nr, navn, aktiv, fødselsdato og alder. De er merket med synlighet.

Det første attributtet har angitt initialverdien "0". Det er også angitt at brukerne anser det for identifikator og den kan derfor verken være null eller tom streng. Det siste er angitt som en restriksjon/egenskapsstreng inne i {klammer}. Navn har ingen initialverdi, men den skal hverken være tom eller null. Tilsvarende er gjort for attributtet *aktiv* som har bare to lovlige verdier og den ene er default. Attributtet alder er avledet og skal ikke lagres men beregnes på forespørsel.

Om operasjonene

Alle operasjonene har her synlighet *public* eller *private*. De operasjonene som har returtype er funksjoner, resten er prosedyrer (returnerer *void* i Java).

De to første operasjonene er konstruktør med argumenter og destruktør. Begge legges til automatisk av kompilatoren hvis man ikke lager dem selv. Det er sjeldent i Java å bruke destruktøren til noe, men her skal tydeligvis noe gjøres.

Deretter følger et antall tilgangsoperasjoner med navn *get...* og *set...* Legg spesielt merke til at attributtet *nr* er deklarert som en String, mens begge tilgangsoperasjonene bruker int. Dette er et eksempel på at informasjonsskjulingen tillater attributter å ha andre typer enn det som vises frem til andre objekter. Videre er det et eksempel på en operasjon som returnerer en verdi som faktisk ikke lagres, nemlig *getAlder*. Fra utsiden er det ikke mulig å vite at det er slik det foregår, men man vil savne *setAlder* og det finner vel de fleste naturlig.

Så følger to operasjoner som både Sun/Oracle og Microsoft anbefaler at man overstyrer, nemlig *toString* og *equals*. Begge arves i flere språk fra toppklassen *Object*, men de arvede operasjonene passer dårlig. Når equals skrives om må også *hashCode* skrives om tilsvarende, fordi det bla er følgende "contract" for de to: Hvis *x.equals(y)* er sann så skal *hashCode(x)* og *hashCode(y)* gi samme tall[15]. Her kan *equals* passende sammenlikne studentenes nr og *hashCode* kan returnere resultatet av en algoritme brukt på nr fordi nr er merket som ID så alle skal være forskjellige[16].

Den siste operasjonen er privat og kan bare brukes av objektet selv. Sannsynligvis skal den konvertere fra int til String og omvendt, kanskje avhengig av hvilken verdi som er oppgitt. Operasjonen tar to argumenter, begge er *inout*, dvs. *ByRef* – alternativene er *in* og *out*. Begge er merket med defaultverdi hvis de ikke har verdi (de kan være null), men de er obligatoriske. Merk spesielt at dette ikke går i Java – jeg har brukt syntaks fra UML/C++/C# – men man kan programmere i Java med tilsvarende effekt og det blir da programmererens utfordring.

Jeg har tidligere nevnt at objekter "vet" hvilken klasse de tilhører, men det lagres ikke i et attributt som man "må huske å ta med". Faktisk har alle objekter flere opplysninger om seg selv enn bare klassetilhørighet.

Som nevnt tidligere kan man abstrahere klassen og bare ta med det som er nødvendig i øyeblikket, f.eks. ikke ta med synlighet, ikke ta med noen attributter og/eller klasser, ikke ta med klassens stereotype osv. Slik ser klassen ut i minimumsform:

Det gir klassediagrammer med lite informasjon, og er derfor ikke vanlig.

Man passer på at objektene har alle attributter som vedrører dem, men heller ikke flere (ref. god liming). Operasjonene er enten lagt her av nødvendighet (konstruktør, destruktør og tilgangsoperasjoner *må* ligge i disse objektene), eller fordi det "passer". Det siste betyr vanligvis at objektet har direkte eller indirekte tilgang til de attributtene som er nødvendige for å svare fornuftig på meldingen.

Det er dessuten vanligvis lurt å delegere mest mulig gjennom meldinger til andre objekter (men se opp for "tramp data"). Hvis man f.eks. har studenter assosiert med kurs og hver student har en liste over "sine" kurs, kan man tenke oss flere måter å finne en students kurs på (med navn, stp osv.). Noen muligheter når A er et objekt som vil lage en streng med studentobjektet Bs nr, navn osv., samt diverse opplysninger om alle Bs kurs:

[15] Spesielt interesserte henvises til http://docs.oracle.com/javase/1.5.0/docs/api/java/lang/Object.html der de kan lese om både equals (hvordan den skal virke) og hashCode.
[16] Man kunne tenkt seg å bare returnere verdien av nr direkte. Da blir jo alle forskjellige. Man vil midlertid gjerne inkludere et primtall i algoritmen fordi hashkodene da fordeles bedre. Det er en fordel i mange sammenhenger.

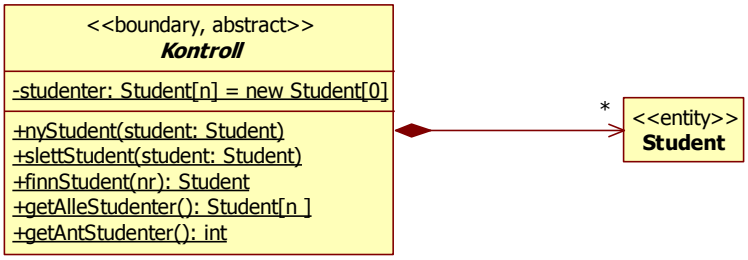
1. A ber studentobjektet B returnere sine attributter inkludert sin samling over kursreferanser. A går selv igjennom samlingen av kurs og ber hvert kurs oppgi sine attributter. Alt dette setter A selv sammen til en fornuftig streng.
2. A ber B om å få Bs toString() og deretter Bs samling av kursreferanser. A går igjennom samlingen og ber hvert kurs returnere sin toString(). A setter alt sammen til én streng.
3. A ber B lage hele strengen. B bruker sin egen toString() og tar deretter selv kontakt med "sine" kurs og ber om å få hvert kurs' attributter. Dem setter B sammen til en passende streng og returnerer.
4. A ber B lage hele strengen. B starter med sin egen toString() og ber deretter hvert av "sine" kurs om toString(). Dette settes sammen og returneres.

Det siste er det beste, fordi det delegerer maksimalt. For å få til dette, må studenten ha en operasjon for det, f.eks. *getStudentMedKurs():String* og de enkelte *toString*-operasjonene må passe (eller spesiallages for dette formålet og ha et annet navn).

Alt ovenfor gjelder også for klasseattributter og klasseoperasjoner, men husk at klasseoperasjoner kun har tilgang til klasseattributtene sine. Andre må hentes med meldinger til objekter og andre klasser. I C++ er det f.eks. ganske vanlig å legge en samling (collection) med klassens objekter som klasseattributt. Klassemetoder gir da tilgang til samlingen, inkludert innsetting og sletting av et objekt. I Java er nok ikke dette vanlig – man legger objektsamlinger utenfor objektenes egen klasse.

Eksempel 2 – med klassemedlemmer

Nedenstående klasse *Kontroll* tar vare på referanse til alle Student-objektene. Siden det ikke er aktuelt med flere objekter av klassen *Kontroll*, legges her alle metodene som klassemedlemmer, og klassen gjøres abstrakt. Dette er altså neppe et Javaprogram, for da ville man nok heller laget en singleton (se neste avsnitt).



Klassenavnet er i kursiv, så dette er en abstrakt klasse. Siden det er vanskelig å skrive kursiv håndskrift, pleier jeg gjerne i tillegg å legge inn {abstract} som restriksjon eller en stereotype som her.

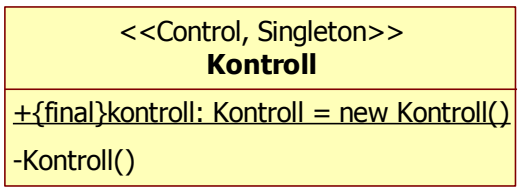
Klassemedlemmer er understreket. Evt. kan man også gjerne gi dem stereotypen <<static>> eller restriksjonen {static}.

"Singletons"

I Java vil man gjerne la data lagres i og arbeidet gjøres av et objekt selv om det bare skal instansieres ett av dem. Slike klasser kalles da "singletons". I praksis kan dette gjøres slik at det lages en klasse som instansierer seg selv med et klasseattributt. Konstruktøren gjøres privat så bare klassen selv kan instansiere, og det gjøres når klassen lastes.

I C++ er det mitt inntrykk at man bare benytter objekter når det skal være mer enn ett objekt av klassen. Istedenfor å instansiere bare ett objekt, bruker man isteden mange klassemedlemmer.

Her er et eksempel på en singleton, der *Kontroll.kontroll* er den eneste mulige instansen av den konkrete klassen *Kontroll:*



Legg merke til at attributtet *kontroll* er et klasseattributt. Det er en konstant (*final*) og kan derfor kun leses. Det er bare klassen selv som kan opprette slike objekter, da konstruktøren merket privat.

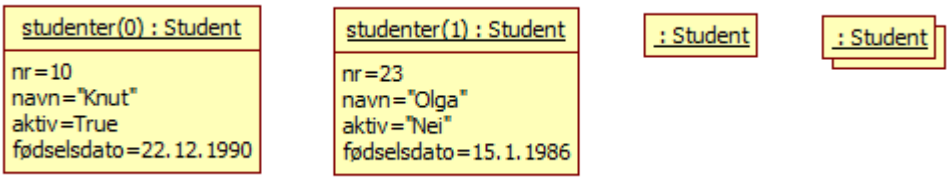
Slik kan det se ut (Java):

```
public class Kontroll { //Singleton
    public static final Kontroll kontroll = new Kontroll();
    private Kontroll(){
    }
}
```

Objektet *kontroll* kalles da med *Kontroll.kontroll*.

Objekter

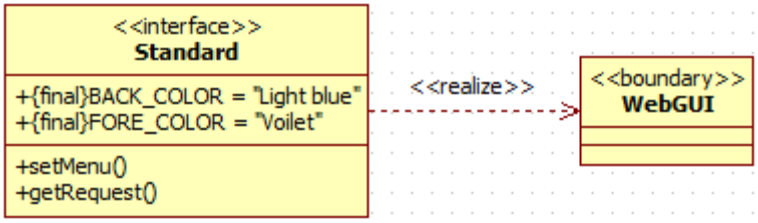
Objekter tegnes sjelden inn i klassediagrammer. Derimot tegnes objekter i andre diagrammer og da ser de slik ut:



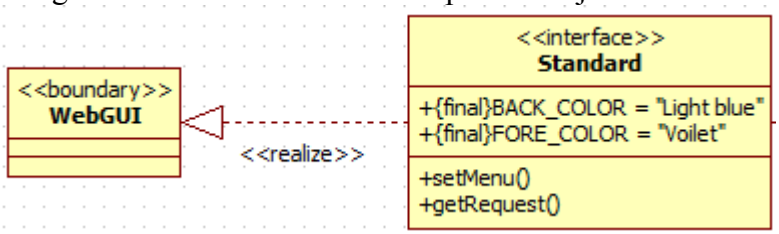
Symbolet er helt likt symbolet for klasser. Forskjellen ligger navnefeltet. Et objekt vil ha kolon foran klassenavnet og foran der kan det stå et navn (det vil si referansen til objektet). Hvis det ikke står noe der betyr symbolet "ett eller annet objekt av denne klassen". Man kan – om ønskelig – fylle ut verdier på attributtene, slik som vist for to av dem her. I de diagrammene de brukes, er dette uvanlig. Det er aldri aktuelt å vise objektets operasjoner – de er like for alle objekter i klassen og vist der. Synlighet vises heller ikke. Symbolet helt til høyre viser "flere objekter av denne klassen".

Grensesnitt

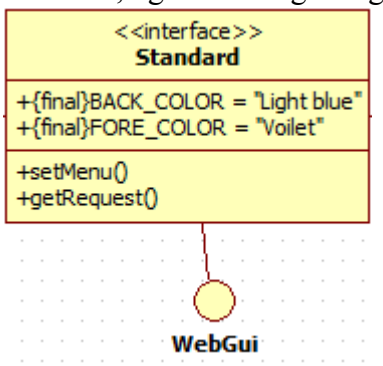
Et grensesnitt i UML og Javaforstand, er i seg selv klasser. De har bare deklarerte men udefinerte operasjoner og/eller konstanter. De er navngitt. De tegnes derfor best i klassediagrammet som en klasse med stereotypen «interface». Realisering kan tegnes som en form for avhengighet:



Alternativt kan det tegnes som en form for arv med prikket linje:



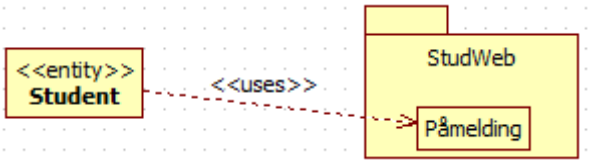
Grensesnitt kan også tegnes som en sirkel, og realiseringen tegnes da som en assosiasjon:



Dette er mer vanlig i andre typer diagrammer.

Pakker

For å abstrahere større diagrammer, kan man tegne *pakker*. Hver pakke inneholder da flere klasser som utgjør en avgrenset del av systemet:



Dette er en abstraksjon. Det er faktisk mulig her å lage et eget diagram for pakken StudWeb og da kan det kanskje ses på som nivåinndeling?

UML spec 2003: Entity, control and boundary classes

4.3.5 Class Stereotypes

4.3.5.1 Entity

Stereotype	Base Class	Parent	Description	Constraints
Entity «entity»	Class	NA	An entity is a passive class; that is, its objects do not initiate interactions on their own. An entity object may participate in many different use case realizations and usually outlives any single interaction.	None.

The notation for Entity is shown below.

4.3.5.2 Control

Stereotype	Base Class	Parent	Description	Constraints
Control «control»	Class	NA	A control is a class whose objects manage interactions between collections of objects. A control class usually has behavior that is specific for one use case, and a control object usually does not outlive the use case realizations in which it participates.	None.

The notation for Control is shown below.

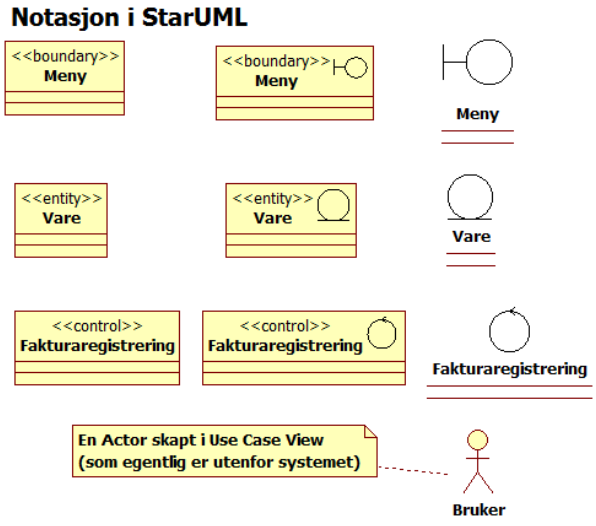
4.3.5.3 Boundary

Stereotype	Base Class	Parent	Description	Constraints
Boundary «boundary»	Class	NA	A boundary is a class that lies on the periphery of a system, but within it. It interacts with actors outside the system as well as with entity, control, and other boundary classes within the system.	None.

The notation for Boundary is shown below.

Kilde: OMG Unified Modeling Language Specification versjon 1.5, 2003

Notasjon i StarUML



Oppgave til kapittel 4 (”Venner”)

Du vil lage et objektorientert system for opplysninger om dine venner (bare navn) og deres telefoner (nr og type – hhv "mobil" og "fast"). Vennens navn og telefonens nummer brukes av deg som ID. Vi antar at hver telefon bare knyttes til én venn, men vennen kan ha flere telefoner.

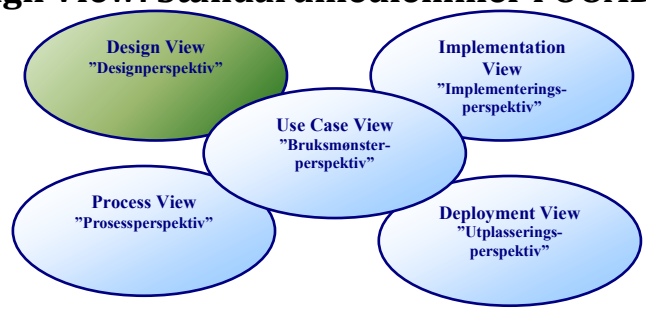
Systemet skal bruke et grafisk brukergrensesnitt, og det skal være mulig å endre, finne, skape og slette en venn (CRUD = create, retrieve, update and delete) og tilsvarende for telefonene. Bruk en egen klasse for GUI og en egen kontrollklasse som styrer alle oppdateringer.

Gjennomfør full informasjonsskjuling. Tenk også igjennom om det kan trengs andre operasjoner enn bare tilgangsoperasjoner.

Du behøver ikke tenke på lagring av objektene.

Oppgave: Bruk StarUML, og tegn klassediagram for systemet.

Kapittel 5 – Design View: Standardmedlemmer i OOAD



Forenkling av analyse og design med standard attributter og standardmedlemmer
Anta at man har en klasse, tegnet slik i klassediagrammene våre

Person (design)
+fornavn:String[20] +etternavn:String[30] ~født:Date +/ alder:int #fnr:long{ID, lovlig personnummer} +adresse:String[30] +postnr:int{norsk postnr 0000..9999} -antTegnINavnet:int {0..50} -nesteFødselsdag: Date{>født}

Spørsmålet er hvordan denne skal realiseres. "Objektorienterte purister" vil hevde at *alle* attributtene bør *realiseres* private, fordi synlige attributter ødelegger innkapslingen. Hvem skal her f.eks. sjekke at fornavn ikke går ut over 20 tegn? At postnr er lovlig? At fnr er lovlig? Skal *alle* kallende objekter gjøre det, eller skal koden legges til klassen selv? Det siste gir betydelig mindre kode, mindre binding, enklere testing og mindre vedlikehold. Men da blir det behov for et betydelig antall operasjoner, fordi private/protected attributter også skal kunne *initieres*[17], *endres* og *avleses*. Man må altså føye til synlige tilgangsoperasjoner:

Person (implementering)
-fornavn:String[20] -etternavn:String[30] -født:Date -fnr:long{ID, lovlig personnummer} -adresse:String[30] -postnr:int{norsk postnr 0000..9999} -antTegnINavnet:int {0..50} -nesteFødselsdag: Date{>født}
+ setFornavn(fornavn:String[20]=""):void + getFornavn():String[20] + setEtternavn(etternavn:String[30]):void + getEtternavn():String[30] ~ getFødt():Date ~ setFødt(fdato:Date):void

[17] Det er ingen implisitt initiering av attributter/objekter i C++ slik det er i Java, men de fleste språk (inkl. Java) kan man legge til en *initializer* i deklarasjonen.

+ getAlder():int ……og tilsvarende for *alle* attributter unntatt de private (avledede skal bare kunne oppgis)

Videre skal man ha konstruktør og kanskje destruktør og overstyre noen arvede operasjoner (typisk *toString*, *equals* og *hashCode*). Alt dette blir svært tungvint og veldig detaljorientert i analyse- og designfasen. Noen forfattere anbefaler derfor at man i en virksomhet eller prosjekt vedtar standarder for hvilke operasjoner som i alle fall skal finnes, slik at man kan konsentrere seg om det som er *uvanlig* og går ut over standarden. Standardoperasjonene spesifiseres da *ikke* i diagrammene. Det er slik jeg vanligvis gjør det.

I nedenstående eksempel er det tre assosiasjoner. De må i praksis realiseres gjennom attributter. All informasjon om dette ligger allerede i diagrammet, unntatt attributtenes navn. De er sjelden kritiske, så her kan programmereren gis stor frihet.

Her må programmereren legge til følgende attributter (jeg bruker her mengder, men det kan gjerne realiseres som en annen type samling):

- ✓ **utleie:** *Leietaker.hytter:Set<Hytte>* og *Hytte.leietakere:Set<Leietaker>*
  Her blir det en samling for begge klasser, da assosiasjonen er mange-til-mange og navigerbar begge veier.
- ✓ **eierskap:** *Hytte.eier:Eier* og *Eier.hytter:Set<Hytte>*
  Her blir det enkel attributt i den ene klassen og samling i den andre, da assosiasjonen er en-til-mange.
- ✓ **telefon:** *Eier.telefoner:Set<Telefon>* og ingen attributter i *Telefon*.
  Her blir det bare attributt i den ene klassen (samling) da assosiasjonen er navigerbar bare én vei.

Da disse attributtene er selvsagte, vil det også forenkle analyse/design om de kan være underforstått. Jeg kaller dem "assosiasjonsattributter" og tar dem sjelden med i diagrammet.

Her er mitt ukomplette forslag til en slik standard. Jeg bruker engelske forstavelser o.l. da det er slik man oftest ser dem i andres programmer og i programmeringsspråkene selv (f.eks. i Java og .NET) der de med et kjælenavn kalles "getter and setter methods". Det er naturligvis ingenting i veien for å bruke norske forstavelser, f.eks. *hentA* isteden for *getA*, *settA* istedenfor *setA* osv.[18]

[18] I løsningsforslag vil du nok finne begge deler (og til dels forskjellige i samme løsning) – det er litt avhengig av dagsformen☺!

Eksempel: En standard

Standard for bedriften "Stan Dard ASA"

Generelt

I dette notatet kalles medlemmene attributter og operasjoner. Det skilles mellom objektmedlemmer og klassemedlemmer ("static").

Det skal gjennomføres lagdeling, så kun grenseklasser kommuniserer med omgivelsene. Tilgangsoperasjonene som *henter* data returnerer data av riktig type. Tilgangsoperasjoner som *endrer* data skal kaste feil hvis endringen ikke ble godtatt.

Om attributtene

Synligheten som er angitt for attributter, skal forstås som synligheten for tilgangsoperasjonene – **attributtene skal uansett realiseres som private**. Det kan gjøres unntak for konstanter og *readonly* attributter som kan ha den angitte synlighet – hensikten er å hindre ukontrollerte endringer av attributtverdiene.

Alle attributter som er realiseres som *private* kan realiseres med en hvilken som helst type. Typeangivelsen *skal* imidlertid brukes for alle synlig operasjoner som henter/endrer attributtverdien.

Attributter som merkes *{ID}* alene eller i kombinasjon, er attributter som brukerne anvender som identifiserende. Programmet skal sikre at det aldri finnes to objekter med samme verdi for slike identifikatorer. De skal behandles særskilt i operasjonene – se nedenfor. Tilsvarende gjelder for attributter merket *{unik}* som angir at det ikke skal fines to like verdier.

Attributter som er merket *{derived}* eller / i kombinasjon med synlighet (-, +, # eller ~), skal realiseres som avledet. De skal da normalt ikke lagres i attributt, hvis ikke de hentes svært mye. Det må da også sikres at de oppdateres ved behov, som f.eks. en persons alder.

Standard operasjoner

Gitt en klasse $Alfa_1$, med synlighet, attributtnavn, attributtets type, evt. initialverdi og domene (som begrenser typen), slik

$Alfa_1$
+,-,#,~,/ a_1:T_1=t_1{$domene_1$}
+,-,#,~,/ a_2:T_2=t_2{$domene_2$}
……
+,-,#,~,/ a_n:T_n=t_n{$domene_n$}

der initialverdi og domene ikke behøver å være oppgitt (da skal hele datatypens domene anvendes og en standard initialverdi brukes). Attributter kan også være merket *{ID}*, *{unik}*, *{NN}*, *{readonly}*, *{const}*, *{static}*, *{derived}* og annet. Flere av disse har konsekvenser for hvilke standardoperasjoner som skal lages og for implementeringen.

Nedenfor angis de operasjonene som er underforstått som standard tilgangsoperasjoner. Dem behøver man ikke å spesifisere under design (klassediagrammer osv.). Andre operasjoner – inkludert andre tilgangsoperasjoner enn de som er angitt nedenfor – må uttrykkelig spesifiseres under design.

Det står programmereren fritt å lage andre (hjelpe-)operasjoner enn dem som er oppgitt, så lenge de merkes *private* så de ikke inngår i klassens grensesnitt.

Operasjoner for enkle attributter
Enkle attributter er enten av primitive typer med bare én verdi (*int*, *boolean*) eller enkeltobjekter (*Date*, *Person*) og altså ikke samlinger.

1) For hvert enkle attributt som ikke er merket privat, lages operasjonen
   +/#/~ getA$_i$():T$_i$
   der synligheten er den samme som for attributtet selv og som returnerer gjeldende attributtverdi.
   Private attributter tillates altså lest direkte internt i objektet/klassen. Det bør allikevel lages *get*-operasjon hvis hentingen innebærer beregninger eller konverteringer.
2) Attributter som er konstanter, *readonly* eller avledet, skal ikke ha endringsoperasjon. Det skal også nøye vurderes om det virkelig er behov for å kunne endre attributter merket *{ID}*.
3) For hvert attributt som skal kunne endres – også private – lages operasjonen
   +/-/#/~ setA$_i$(a$_i$:T$_i$)
   der synligheten er den samme som angitt for attributtet. Operasjonen kontrollerer argumentet og
   a) gir attributtet den angitte verdien og returnerer uten feil, eller
   b) kaster feil uten å endre attributtverdien

Samlinger
Med *samling* forstår man her flere verdier eller flere objekter samlet i ett attributt, f.eks. lister, en trestruktur.

Gitt en klasse *Alfa$_2$*, slik

Alfa$_2$
+,-,#,~ a$_1$:samling$_1$(T$_1${domene})=t$_1$ +,-,#,~ a$_2$:samling$_2$(T$_2${domene})=t$_2$ …… +,-,#,~ a$_n$:samling$_n$(T$_n${domene})=t$_n$

der domenet *samling(T$_i$)* er en samling av objekter av den underliggende typen *T$_i$*. Initialverdi og domene ikke behøver å være oppgitt (da skal hele datatypens domene anvendes og *a$_i$* initieres til en tom samling – for arrays settes hvert element til *null*).

For hvert slikt attributt a$_i$ lages følgende operasjoner, der synligheten er som for attributtet selv:
1) Avledede attributter skal ikke realiseres, men gir bare utgangspunkt for tilgangsoperasjoner som returnerer verdier.
2) For hver samling som ikke er merket privat, lages operasjonene *+,#,~ getA$_i$():samling$_i$(T$_i$)* der synligheten er den samme som for attributtet selv og som returnerer hele samlingen. Private attributter tillates altså lest direkte internt i klassen/objektet.
   Det bør vurderes å returnere andre typer samlinger enn den som attributtet egentlig er deklarert med, f.eks. kan det være aktuelt å returnere en samling både som liste og trestruktur.

3) Hvis den underliggende typen er en klasse, skal det vurderes å lage en operasjon med slik signatur:
+,#,~ findA$_i$({opt}para$_1$:type$_1$, {opt}para$_2$:type$_2$...{opt}para$_n$:type$_n$):samling$_i$(T$_i$)
der *para$_1$:type$_1$* osv. er ønskede verdier for attributter i de underliggende objektene og har samme type som dem. Alle attributtene i den underliggende klassen skal være med, men det skal være frivillig å oppgi dem[19]. Hvis et parameter ikke er oppgitt som aktuelt argument i kallet, skal det bety at det ikke stilles krav til denne verdien. Operasjonen returnerer alle objektene som har de ønskede verdiene, evt. en tom samling.
4) For hver samling som ikke er merket privat og som ikke er avledet, lages operasjonene
   a) *+,#,~ addA$_i$(a$_i$:T$_i$)* som legger en ny verdi/objekt *a$_i$* inn i samlingen.
   b) *+,#,~ delA$_i$(a$_i$:T$_i$)* som fjerner verdien/objektet a$_i$ fra samlingen.
5) Det bør vurderes å ta med en operasjon som returnerer antall objekter i samlingen:
+,#,~ countA$_i$():int
For arrays bør dette være antall elementer som faktisk er i bruk, og ikke arrayens størrelse.
6) Avhengig av systemets behov, kan det være aktuelt å lage operasjoner som sorterer elementene i samlingen, fjerner dubletter, teller antallet unike verdier og annet. Standarden stiller ingen krav om slike operasjoner.
7) Samlinger kan merkes *{ID}* alene eller i kombinasjon med andre attributter. Dette antas å være sjelden. For slike samlinger der den underliggende typen er en klasse, skal det lages en operasjon *+,#,~ findA$_i$(para$_1$:type$_1$, para$_2$:type$_2$...):T$_i$* der parametrene er ønsket verdi for et/flere ID-attributt i de underliggende objektene og har samme type som dem. De utgjør altså verdien på ID-attributt for det ønskede objektet og returnerer bare ett objekt evt. *null*.

Overskriving av arvede operasjoner
Man skal vurdere å overskrive alle arvede operasjoner med sikte på polymorf oppførsel. Uansett skal nedenstående operasjoner overstyres for alle klasser, f.eks. klassen *Alfa*:
1) *toString():String* som gir en representasjon av objektets attributtverdier i én streng
2) *equals(o:Object):boolean* eller tilsvarende som returnerer *True* hvis objektet selv oppfattes som "likt" med det oppgitte objektet, basert på verdier av ett/flere attributt(er) og at de er av samme klasse.
3) *hashCode():int* eller tilsvarende som returnerer et unikt heltall for hvert objekt.
4) Det skal også vurderes å implementere alle sammenlikningsoperatorer[20] for klassen. Hvis det ikke er mulig (Java), må man da isteden overstyre *compareTo(o:Object):int*.

Konstruktør/destruktør
1) Alle klasser (f.eks. *Alfa*) skal realiseres med minst én standardkonstruktør
+,#,~ Alfa(a$_1$:T$_1$=t$_1$,a$_2$:T$_2$=t$_{2,...}$a$_n$:T$_n$=t$_n$).
Følgende argumenter skal være med i argumentlisten:
   a) Attributter merket *{ID}*
   b) Attributter som ikke får være *null* eller ha en standardverdi
Dette gjelder bare attributter med minst samme synlighet som konstruktøren.
Konstruktøren skal uansett initiere *alle* attributter, enten (i prioritert rekkefølge)

[19] Dette er ikke så enkelt i Java, som ikke støtter valgfrie parametre, men en måte er å tillate null som parameterverdi. Det vil imidlertid skape problemer for enkle datatyper. En løsning er å bruke strenger (som *kan* være null). Mange Java-programmerere anbefaler isteden å lage flere operasjoner med varierende kombinasjoner av attributter.
[20] Det minnes om at = = og != sammenlikner *referanser*, mens compareTo() og equals() sammenlikner *attributtverdiene* (objektets tilstand).

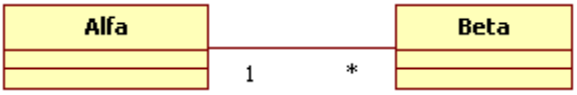
c) til den argumentverdien som er oppgitt i kallet, eller
d) til den angitte initialverdien, eller
e) til standard initialverdi for typen

Konstruktøren skal kontrollere at domener overholdes. Hvis minst én argumentverdi er feil, skal det kastes feil.

2) Det skal vurderes å lage flere konstruktører med varierende antall argumenter (de samme som i punkt 1 men med tillegg av flere, evt. kan de være *optional* (ikke Java)).
3) Det skal vurderes å lage/overstyre en destruktør[21] som frigir eksterne ressurser som objektet har bundet. Andre destruktører skal ikke lages.

Singletons
Klasser som er merket <<*Singleton*>> skal realiseres slik at det kun kan skapes ett objekt av klassen.

Assosiasjoner
Alle assosiasjoner som er navigerbare, skal realiseres som attributter. Hvis ikke annet er angitt, anses assosiasjonen å ha synlighet public. Attributtene skal likevel realiseres private med tilgangsoperasjoner som angitt ovenfor (”Enkle attributter” og ”Komplekse attributter”).



I eksemplet ovenfor innebærer det at klassen *Alfa* skal ha et komplekst attributt *+beta:Samling(Beta*) og Beta skal ha et enkelt attributt *+alfa:Alfa*, begge med tilsvarende tilgangsoperasjoner. Merk at *beta* kan være en tom samling, mens *alfa* ikke kan være *null*. Dette får konsekvenser ved instansiering og sletting – programmereren må påse at reglene overholdes.

Hvis samlingstypen er angitt, skal den brukes om mulig. Hvis samlingen er oppgitt som *Set*, *mengde* eller lignende skal programmereren sikre at det ikke legges inn to like objekter/verdier.

Spesielt for persistente klasser
For persistente klasser trengs det i tillegg et apparat for å lagre/hente objektene til/fra ytre lager. Det står programmereren fritt å implementere persistensen på de måter som virksomheten godtar.

Normalt legges persistensen i en kontrollklasse som gjennomfører tilstandsendringer, overensstemmende med den lagdelingen som er nevnt helt først under "Generelt". Alternativt anvender noen egne grenseklasser (oftest Singleton) til dette

Slutt på eksempelet for en standard.

[21] I Java har alle klasser automatisk destruktøren *finalize()*, som kan overstyres. Den anvendes av ”garbage collector” som dessverre går i egen tråd og ikke kan styres i særlig grad. Det advares sterkt mot å kalle den selv. Isteden anbefales å lage en egen operasjon *delete()* som foretar nødvendig opprydding før objektet derefereres (så ingen variable lenger referer til det).

Valgets kval: Domain driven eller function driven design?

Spørsmålet i overskriften kan omformuleres til et spørsmål om man bør ta utgangspunkt i entitetsklassene ("Entity driven") eller i bruksmønstrene ("use case driven") når man lager designet.

Bakgrunn "fra gamle dager"

Når man skal designe et system, kan man velge forskjellige strategier. I god, gammeldags imperativ programmering[22] er det et klart skille mellom data og funksjonalitet – bl.a. blir de deklarert og oppbevart hver for seg. Da valgte man gjerne mellom designstrategiene "top down" og "bottom up", eller mellom "inside out" og "outside in"[23]. Noen kjekkaser påsto også at det beste var "any way it works", men de valgte vel da egentlig ingen strategisk innfallsvinkel.

Valgets konsekvenser

I objektorientert programmering samler man data og funksjonalitet i klasser/objekter. Det reiser spørsmålet om beste strategi:

1. Bør man begynne med funksjonaliteten (operasjonene), samle dem i klasser og så legge dataene i de klassene der operasjonene trenger dem = *funksjonsdevet design* (UML: "use case driven").
2. Bør man begynne med dataene (attributtene), samle dem logisk i klasser og så legge operasjonene der det finnes data som operasjonene trenger = *entitetsdrevet design* (UML: "entity class driven")

Man kan forvente at funksjonsdrevet design vil gi mer logisk og kanskje raskere funksjonalitet. Samarbeidet mellom objektene skal føre til at systemet gjør noe nyttig, og det blir enklere og bedre. Bl.a. kan man legge operasjoner som samarbeider tett i samme klasse. Attributtene innføres av hensyn til operasjonene – de skal ta vare på tilstand. Derved blir det lite sending av meldinger (kallene foretas internt i objektene). På den annen side vil sammenhengen mellom attributtene bli dårligere, eller til og med redundant. Kanskje passer dette best hvis systemet skal modellere en verden med lite data og mye oppførsel, f.eks. et prosesstyringssystem der objektene representerer maskiner, kjemisk prosesser o.l. Brukeren vil ha hjelp til å styre noe. Jeg ser f.eks. for meg styring av jernbanelinjer med trafikklys og penser eller regulering av fly i et luftrom. Dataene om hvert tog/fly er få, men det er svært viktig – helst i real time – å modellere hva de *gjør*. I slike systemer blir gjerne kontrollklassene de viktigste, mens entitetsklassene og grenseklassene får mindre betydning og er enklere å modellere.

---

[22] Det skilles ofte mellom fem hovedparadigmer i programmering:

- Det *deklarative* paradigme: Beskriver resultat – kompilatoren bestemmer hvordan det skal finnes som i SQL.
- Det *imperative* paradigme: Gjør først dette, så dette (sekvensielt), som i Fortran, Algol, C, Pascal, Basic
- Det *funksjonelle* paradigme: Evaluer dette uttrykket og bruk resultatet til noe, som i matematiske applikasjoner som f.eks. i Scheme, Lisp der alt er faste definisjoner (ingen variable, ingen kall).
- Det *logiske* paradigme: Svarer på et spørsmål gjennom søk etter en løsning ved å kombinere tautologier (sannheter, påstander), f.eks. Prolog
- Det *objektorienterte* paradigme: Send meldinger mellom objekter som en simulering av utviklingen over tid i den virkelige verden, som i Java, C++/C#, SmallTalk, Visual Basic osv.

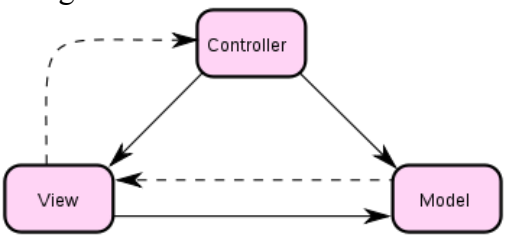
[23] "Top down" startet med overordnede metoder (funksjoner/prosedyrer) og realiserte først underordnede metoder som stubber (så programmet kompilerte). Deretter fylte man på med å kode de underordnede metodene eller hentet dem fra et bibliotek osv. til alt virket. I "bottom up" startet man med de underordnede metodene "som det sikkert ble bruk for" og satte dem så senere sammen med overordnede metoder som brukte dem til et ferdig program. Når man lager et bibliotek, kan man godt si at man legger opp til "bottom up". "Inside out" starter med sentrale "kjernemetoder" og arbeider ut mot grensesnittet, mens "outside in" gjør motsatt.

Man kan forvente at entitetsdrevet design gir bedre sammenheng mellom dataene, mens operasjonene blir plassert "litt her og litt der" avhengig av hvor dataene finnes som de trenger. Starten av dette arbeidet vil likne svært på datamodellering for databaser – man søker entitetsklasser og attributter til hver klasse etter hva systemet må ta vare på. Deretter må man ha mange operasjoner som håndterer dataene (tilgangsoperasjoner) noen "obligatorisk operasjoner" (konstruktør/destruktør, toString og equals). Når alt dette er på plass, kan man se hvor man må plassere operasjoner som gjør noe virkelig nyttig for brukerne, dvs. man må realisere bruksmønstrene. Da vil hver funksjonalitet føre til mange operasjoner i forskjellig klasser og det må sendes mange meldinger mellom objektene. I slike systemer er entitetsklassene og grensesklassene gjerne de mest sentrale. Dette kan særlig passe for "administrative systemer" der brukerne vil at systemet først og fremst skal ta vare på dataene deres, f.eks. data om kunder, varer, ordre/salg/forsendelse, faktura, ansatte osv. Funksjonaliteten dreier seg om å endre, sette inn, slette og finne data. Det er mange data om en kunde/vare, men de *gjør* ikke så mye og det er ikke så viktig at modellen er helt à jour til enhver tid.

Valget mellom de to kan også avgjøres av hva som er mest "stabilt" over tid. Det er viktig at vedlikehold konsentreres om få klasser.

Model-View-Controller[24] (MVC)

MVC er en arkitektur med tre lag:



MVC med tre lag som kommuniserer.
Heltrukne linjer er direkte forbindelser, prikkede er indirekte f.eks. via en lytter

Model inneholder objektene som modellerer domenet. Det svarer på henvendelser om tilstand og om endringer – vanligvis fra *Controller*. *View* viser modellen i en form som egner seg for interaksjon med andre i systemomgivelsene, typisk med brukere, og håndterer hendelser som brukeren initierer (som klikk på en knapp). Kontrollen mottar hendelsene og input fra viewet og omgjør det til handlinger i modellen.

Domain Driven Design (DDD)[25]

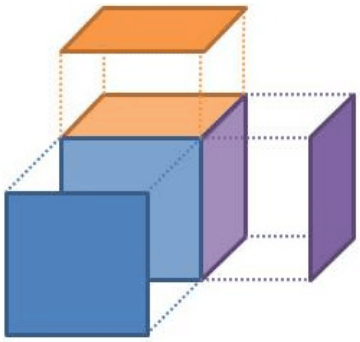
DDD kan sees på som en mer formell og videreutviklet entitetsdrevet strategi og har hentet mange ideer også fra datamodellering. Grunnlaget er at når systemene blir komplekse, er det viktig at de er basert på en modell.

Primært ønsker man å bygge en modell av den virkeligheten som systemet skal operere i, en modell av den virksomheten som skal ha systemet. Det som er spesielt for denne virksomheten ("domain") er i fokus, og modellen av det utgjør kjernen i det nye systemet. Dette anses viktigst, da det er det som skiller en virksomhet fra en annen. Hvis oppgaven som

[24] Først beskrevet av Trygve Reenskaug, 1979, da han jobbet med SmallTalk hos Xerox Parc.
[25] Eric Evans, 2004, i boken "Domain Driven Design". Det jeg skriver om her er i stor grad – inkludert figurer – hentet fra http://www.methodsandtools.com/archive/archive.php?id=97.

skal løses er lik andre virksomheters, bør man vurdere et standardsystem i stedet. Persistens og grensesnitt legges til senere.

"Modell" i DDD betyr koden. Koden kan representeres i diagrammer og de er da "views" av modellen, men det til syvende og sist er det koden som er selve modellen.



Modellen er i sentrum, flatene rundt representerer forskjellige "views".

For å kunne diskutere modellen fornuftig, må alle bruke samme språk, dvs. ord og uttrykk fra domenet. Det kalles "ubiquitous language" dvs. " det allestedsnærværende språket". Grupper av mennesker og/eller systemer i omgivelsene utgjør "bounded context" (BC) i omgivelsene, og det er deres språk som benyttes. Det er ikke likt for alle BC, f.eks. vil et databasesystem ha andre begreper enn en sluttbruker.

DDD er mest opptatt av kjernen i modellen, men den forutsetter at andre lag eksisterer:

Lagene i DDD. Domain layer er det sentrale, men de andre må også lages

Presentasjonslaget er det omgivelsen "ser" – man kaller det grenseklasser. Applikasjonslaget "får jobbene gjort" (ofte kalt "business logic" eller "logisk lag") og holder orden på hvor langt en forespørsel er kommet. Man kaller det kontrollklasser. Domenelaget inneholder tilstanden til de modellerte entitetene og utgjøres av entitetsklasser. Infrastrukturlaget tilbyr vårt system tjenester/hjelpefunksjoner og gjennomfører persistens.

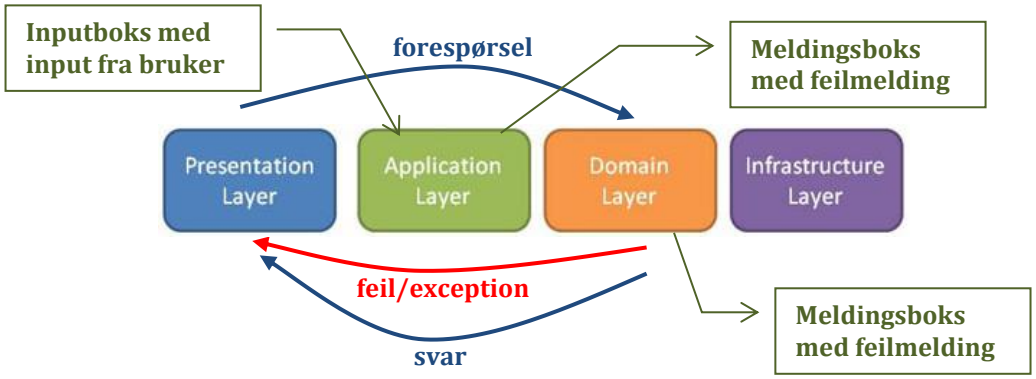
Det sentrale i DDD blir da at entitetsklassene blir svært viktige.

Entitets- eller funksjonsdrevet design?

I MVC er det ikke spesielt svart på dette. DDD mener definitivt at systemutviklingen bør begynne med entitetene – funksjonene henges på senere.

Selv vil jeg nok – med referanse til DDDs firelagsmodell, helle til at det er nyttig – for virksomhetssystemer – å begynne med entitetsdrevet design av systemets kjerne. Deretter kan det være nyttig å drive funksjonsdrevet design for kontrollklasser, og tilslutt det samme for grenseklasser. Jeg begrunner det med at enitetsklassene representerer det mest stabile i systemet. For at systemet skal være fleksibelt, er det svært viktig at det er godt modellert. Jo nærmere man kommer omgivelsene, desto viktigere blir funksjonene og klassene blir også mer ustabile pga endringer – "turbulens" – i omgivelsene, f.eks. når GUI eller databasen endres, ny teknologi skal implementeres osv.

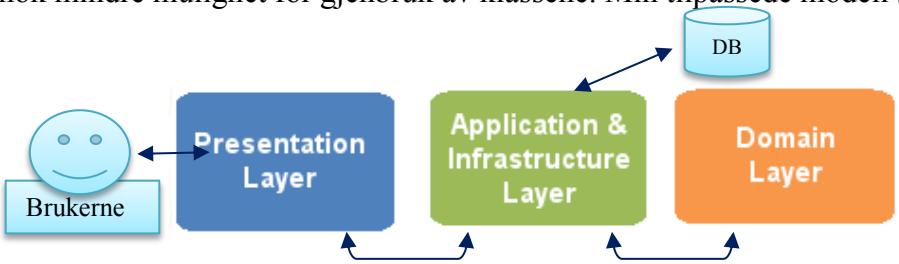
For "administrative systemer" starter jeg følgelig alltid med å analysere/designe entitetsklassene, resten kommer senere. Jeg er svært nøye med at bare grenseklassene kommuniserer med omverdenen. En entitetsklasse får altså ikke lov til å kommunisere en feil til omgivelsene. Slike feil skal kastes (evt. kommuniseres tilbake som spesielle returverdier) og andre objekter nærmere systemgrensen skal håndtere dem. Kanskje kan kontrollklassen løse problemet, men hvis omgivelsene skal ha beskjed, skal meldingen genereres av en grenseklasse. Tilsvarende får en grenseklasse ikke lov til å kommunisere direkte med entitetsklasser – henvendelser og svar skal gå via en kontrollklasse. Min egen strategi er altså å holde lagene klart adskilt, så da vil dette *ikke* være tillatt:



Fyfy! Meldinger som bryter skillet mellom lagene og feil kommunikasjon med brukeren

Jeg kan nok ikke påstå at jeg *aldri* programmerer slik – særlig i begynnerundervisningen – men jeg kan ærlig si at jeg ikke *ønsker* å programmere slik☺!

En endring av modellen som jeg gjør jevnlig, er å slå sammen infrastrukturlaget med applikasjonslaget. Jeg begrunner det med at alle henvendelser fra brukere evt. andre systemer om tilstandsendringer i domenelaget må gå igjennom applikasjonslaget. Objektene der "vet" altså om alle tilstandsendringer. Da er det praktisk at objektene i applikasjonslaget også sikrer persistens av endringene. Alternativt må det sendes mange meldinger fra objekter i domenelaget til objekter i infrastrukturlaget for å få det til. Det fører til duplisering av kode ved at alle klassene i domenelaget må ha kode for å sikre sin egen persistens. En slik sammenslåing som jeg her foreslår gir ingen ekstra kode eller redundans, men det blir da riktignok mindre mulighet for gjenbruk av klassene. Min tilpassede modell ser da slik ut:

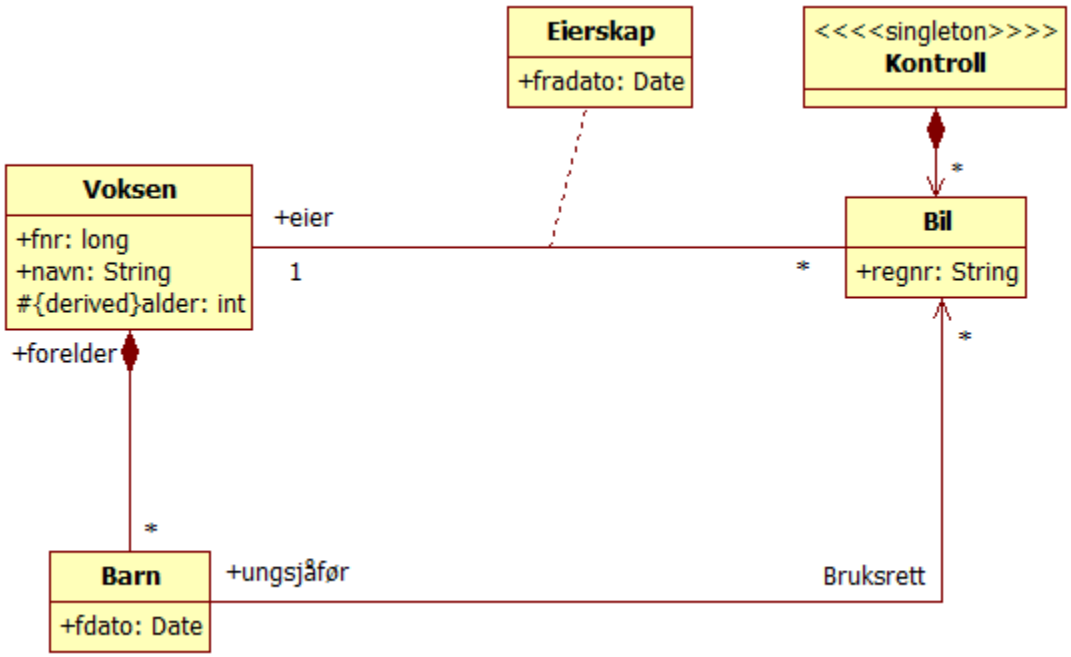


Min tilpasning av DDD

Feil skal håndteres så snart det er mulig – feil som ikke kan håndteres kastes videre inntil de evt. når presentasjonslaget som varsler brukeren om feilen. Da blir det opp til brukeren å stå for korrektive tiltak.

Oppgave til kapittel 5 (standardmetoder)

Nedenfor er det gjengitt et klassediagram.



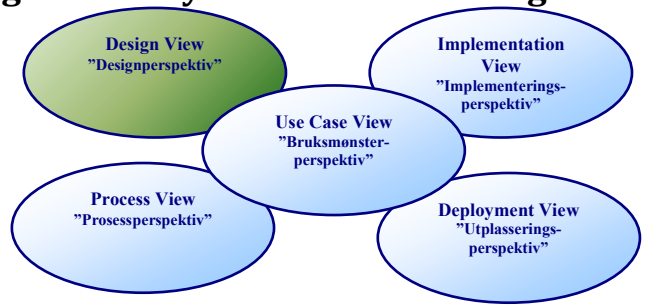
Alle klassene unntatt *Kontroll* er persistente og instansene skal lagres i en relasjonsdatabase. Alle klassene må følgelig ha en unik ID. Angi hvilke det er med egenskapsstrengen *{ID}*.

Du behøver ikke legge til grenseklasser som håndterer brukerne.

Oppgave A: Gjør om på diagrammet så det viser alle de attributter og metoder som skal realiseres ifølge Knuts forslag til standardmetoder, med den synligheten som skal realiseres.

Oppgave B: Vurder om det vil være mulig å få tilgang til alle objektene i systemet ved hjelp av klassen *Kontroll*.

Kapittel 6 – Design View: Dynamisk modellering av enkeltobjekter



Med klassediagrammene dokumenterer man det *statiske*, dvs. klasser/objekter som skal lages, hierarkier, sammenhenger osv. Ethvert system har også *dynamikk* – noe skjer. UML kaller det for "oppførsel". Det må også beskrives.

Dokumentasjon av oppførsel - oversikt

I OOP har alle objekter

1) identitet
2) tilstand
3) oppførsel

Dette gjelder også for *klasser* med klasseattributter og klasseoperasjoner. For å forenkle bruker jeg nedenfor bare ordet objekt.

Mye oppførsel er opplagt og behøver ikke dokumenteres. F.eks. er det opplagt hva et objekt som har attributtet *–navn:String {NN, ikke tom streng}* skal gjøre i funksjonen *+getNavn():String*. Det er også opplagt at *+setNavn(navn:String)* da må kontrollere at parameteret ikke er *null* eller en tom streng før den tilordner attributtet ny verdi. At den skal kaste feil ligger sikkert i en eller annen standard. Hvis den skal gjøre noe mer, må det dokumenteres. Videre må samarbeid mellom objekter og evt. tilstander som objektene gjennomgår, dokumenteres. De samarbeider ved å sende *meldinger* og *signaler* til hverandre.

UML tilbyr følgende dokumentasjonsformer:

1. **Tilstandsdiagrammer** som beskriver hvilken "livssyklus" objektet gjennomgår, og hvordan objektet reagerer forskjellig på de samme meldingene avhengig av hvilken fase i livssyklusen de er. Fokus er på hendelser som resulterer i nye tilstander.
2. **Aktivitetsdiagrammer** viser handlingene som utføres og rekkefølge/logikk. Det minner sterkt om tilstandsdiagrammer, men har fokus på aktiviteter (derav navnet) – tilstandsovergangene skjer implisitt når aktiviteten er slutt. Tilstanden vedvarer altså her mens noe gjøres.
3. **Sekvensdiagrammer** viser hvordan objektene samarbeider gjennom å sende *meldinger* og *signaler* til hverandre for å få til en bestemt funksjonalitet. Fokus er på rekkefølgen av meldingene/signalene (derav navnet) over tid. Det fremgår hvilke objekter som samarbeider, men det er meldingene/signalene og rekkefølgen som er sentrale.
4. **Samarbeidsdiagrammer** viser også hvordan objektene samarbeider gjennom å sende meldinger/signaler til hverandre. Fokus er her på hvilke objekter som samarbeider med hvilke. Rekkefølgen av meldingene vises bare gjennom nummerering, så den er vanskeligere å se.
5. **Signaler** har egne symboler og dokumenteres i særskilte diagrammer.

Dokumentasjon av enkeltobjekters oppførsel

I UML dokumenteres hvert objekts oppførsel med to, forskjellige diagrammer:

1) Statechart Diagram (tilstandsdiagram)
2) Activity Diagram (aktivitetsdiagram) – også kalt ”activity graph”

Disse brukes litt forskjellig. Tilstandsdiagrammer brukes for å vise objektenes tilstander, og hvilke hendelser (stimuli i form av meldinger/signaler) som fører til overgang fra én tilstand til en annen. Tilstandsdiagrammene benyttes først og fremst som beskrivelse for instanser av en klasse (altså objekter av denne klassen). Aktivitetsdiagram er en variant av tilstandsdiagrammer. Tilstandene er her aktiviteter som tar en viss tid og aktivitetsdiagrammer brukes primært for å dokumentere operasjoner (og andre typer handlinger). Aktivitetsdiagrammene likner svært på det som ellers er kalt flow chart (flytdiagram).

Aktivitetsdiagrammer drives primært av at en intern, synkron aktivitet er avsluttet og viser således en *flyt*, mens tilstandsdiagrammer drives av eksterne, asynkrone hendelser (gjerne mottakelsen av en melding/signal).

UML beskriver dette slik:

> **Statechart diagrams** represent the behavior of entities capable of dynamic behavior by specifying its response to the receipt of event instances. Typically, it is used for describing the behavior of class instances, but statecharts may also describe the behavior of other entities such as use-cases, actors, subsystems, operations, or methods.
>
> An **activity graph** is a variation of a state machine in which the states represent the performance of actions or subactivities and the transitions are triggered by the completion of the actions or subactivities. It represents a state machine of a computation itself.
>
> *Kilde: OMG Unified Modeling Language Specification versjon 1.5, 2003*

Tilstandsdiagram (State Machine Diagram)

Tilstanden defineres her som ***verdien av alle attributtene sett under ett***. Når én attributtverdi endres, er også objektets tilstand endret.

> The ***state*** of an object encompasses all of the attributes of the object plus the current data values assigned to the variables that implement the attributes. When we talk about the current state of the object, we are really talking about the current values of its variables. When we say that an object has changed state we really mean that one or more variable values have changed.
>
> *Kilde: http://www4.desales.edu/~dlm1/it532/class07/objattr.html*

I databaser bruker man tilstand om entitetene (representert ved en rad i en/flere tabeller) i denne betydningen.

I tillegg vil mange mene at tilstanden skal inkludere ***verdien av programtelleren***, dvs. hvor langt objektet er kommet med en oppgave den er i ferd med å utføre. I praksis vil man ikke være opptatt av programtelleren, men heller se på tilstanden når objektet "er i ro" og venter. Tilstandene man her diskuterer vil altså vare en stund.

UML skiller ikke særlig skarpt mellom på den ene side *meldinger* som sender data, og på den annen side *signaler* som er uten data men signaliserer at noe har skjedd. Samlet kaller UML dem for *enkelthendelser* (*event instances*). Nedenfor kaller jeg det bare *hendelser*. Når en hendelse ankommer til et objekt, sies objektet å motta en *stimulus*.

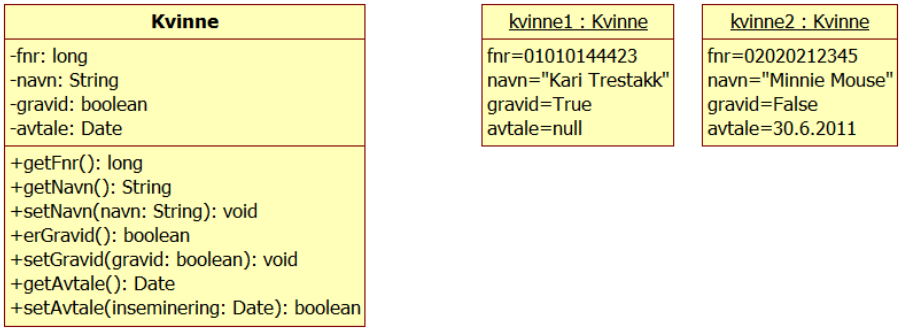
Oppførselen til et objekt styres av objektets tilstand, dvs. hvor langt objektet er kommet i sitt "livsløp"[26]. Tilstanden påvirker objektets oppførsel. Endringen er synlig gjennom forskjellig

[26] Ordet "livsløp" kan gi assosiasjoner i retning av en sekvens som fødsel, oppvekst, voksen, død. Det er ikke meningen. "Livsløpet" kan i prinsippet vare evig – det kan gå fra tilstand til tilstand i en ring, et nettverk e. a.

respons på samme *stimulus*. Objektene får stimuli i form av *hendelser* (*meldinger* og *signaler* som de mottar) og responsen er synlig som en *reaksjon* på meldingen/signalet. Reaksjonen er å gjøre noe, f.eks. å endre egne attributtverdier eller sende melding/signal til andre objekter.

En tilstand er varig. Sålenge objektet er i denne tilstanden, vedvarer oppførselsmønsteret. Tilstanden kan endres, og en slik endring anses *atomær*, dvs. at mens endringen pågår, er det ikke mulig å vite hvilken tilstand objektet er i. Objektet går altså fra tilstand til tilstand, og det finnes ingen "mellomtilstander". (En kvinne kan ikke være "litt gravid".) Tilstander er således *diskrete*. Objekter vil alltid ha en oppførsel, og de er altså alltid i en eller annen tilstand. Hvis man vil holde orden på dem, må man liste opp *alle* tilstander som objektet kan ha.

For å vite hvilken fase av livsløpet et objekt er i, vil det normalt være et attributt (eller kombinasjon av attributter) som lagrer det. Dette er altså spesielle attributter, hvis verdi i høy grad påvirker oppførselen, langt mer enn verdien av *navn*, *fødselsdato* osv. De endrer ikke bare oppførselen ved å returnere en annen verdi, men påvirker det *generelle* oppførselsmønsteret. Et eksempel:



I denne klassen er det attributtet *gravid* som angir tilstanden. Med operasjonen *erGravid()* kan andre objekter "lese av" tilstanden og *setGravid()* endrer den[27].

I figuren er *kvinne1* og *kvinne2* objekter av klassen *Kvinne*. Som man ser er *kvinne1* gravid, men ikke *kvinne2*. Man vil forvente at *kvinne1* vil reagere annerledes på anmodning om å bekrefte en avtale om kunstig befruktning: Hun vil avvise hele forespørselen (f.eks. gjennom å kaste en feil), mens *kvinne2* isteden vil sjekke om datoen passer og evt. notere seg den (responsen blir "ja", "nei"). De oppfører seg altså forskjellig etter mottakelsen av samme stimulus (= samme hendelse) fordi de er i forskjellig tilstand.

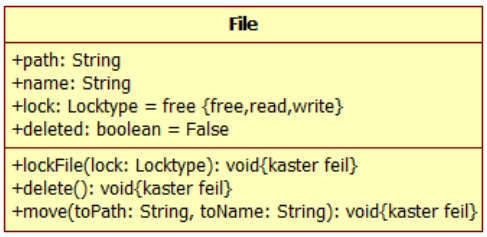
Riktignok endres et objekts tilstand hver gang en attributtverdi endres. Allikevel vil man ikke dokumentere alle tilstandsendringer. F.eks. vil meldingen *kvinne1.getFnr()* gi et annet svar enn *kvinne2.getFnr()*, men det dreier seg jo bare om at *verdien som returneres* er ulik – *oppførselen* er den samme. Man vil da *generisk* angi at *getFnr()* returnerer verdien på attributtet *fnr*. Da vil ikke *det generelle oppførselsmønsteret* endres seg selvom objekt endrer *tilstand* gjennom endringen av attributtet *fnr*. Ytterligere dokumentasjon blir da unødvendig. For at man skal "gidde" å definere forskjellige tilstander, må altså selve *oppførselsmønsteret* være forskjellig. Det er slike tilfeller som omtales her.

Noen reaksjoner styres av operasjonenes parametre. F.eks. kan *setNavn(navn:String)* kaste feil hvis det oppgitte navnet er ulovlig, eller det kan endre verdien av attributtet *navn*. Det har ingenting med objektets *tilstand* å gjøre, men styres av argumentverdien. Objektet vil reagere

[27] Det er enklere å bli gravid i en modell enn det ofte er i virkeligheten☺!

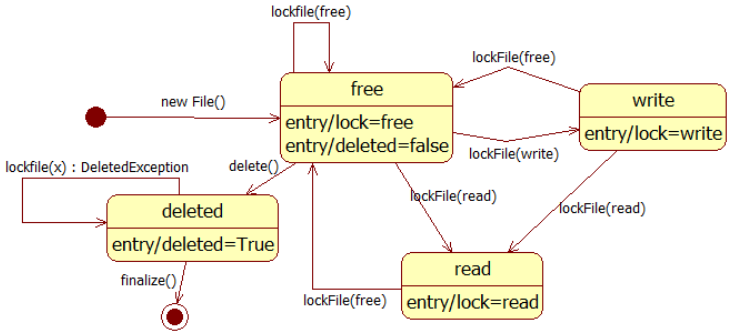
likt på samme melding med samme parameter en annen gang. Det er følgelig utenfor denne diskusjonen. Det man ser på her, er isteden at et objekt også kan reagere forskjellig fra gang til gang, selvom stimulus (meldingen/signalet) er nøyaktig den samme. Da vil *setNavn("Knut")* noen ganger gi én reaksjon, andre ganger en annen reaksjon.

Eksemplet nedenfor er basert på følgende klasse:



Som det fremgår, er det her to typiske tilstandsattributter, nemlig *lock* og *deleted*.

Tilstandsdiagram som dokumenterer tilstandene kan se slik ut:



Tilstandene er tegnet inn som avrundede rektangler, og det er fire av dem. I tillegg er den en start og en slutt. Tilstanden ”free” er den tilstanden objektet starter i, og kalles *initialtilstand* (eller start-tilstand). Den inntreffer med *new File()* og det forutsettes at alle attributter settes til defaultverdi. Tilstanden ”deleted” er alltid den siste tilstanden dette objektet er i, og kalles derfor *terminaltilstand* (eller slutt-tilstand). Her angir det at objektet som representerer filen er merket som slettet, men objektet finnes fortsatt.

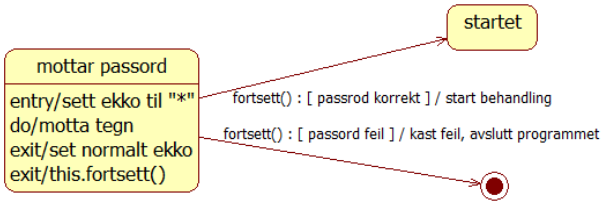
De fleste tilstander kan objektet både komme til og forlate – de er følgelig *transiente tilstander*, mens ”deleted” er permanent – den er følgelig *persistent*. Så vidt jeg kan forstå (jeg kan i alle fall ikke komme på noe moteksempel) må alle terminaltilstander være persistente og omvendt. Legg merke til at i eksemplet kan objektet fortsatt reagere når det er i tilstand ”deleted” f.eks. ved å kaste feil. I andre tilfeller kan det reagere på mange måter – det kan bare ikke komme i ny tilstand.

Den store prikken angir at objektet skapes, mens prikken med ring rundt angir at det slettes. Før det er skapt og etter at det er slettet, finnes ikke objektet og da kan det naturligvis ikke ha noen oppførsel. (I eksemplet betyr det at objektet faktisk er fjernet fra systemet eller i det minste dereferert og ikke lenger tilgjengelig.)

For hver tilstandsovergang, er det angitt den hendelsen som fører til overgangen, i form av en melding/signal som objektet mottar. Hendelsen beskrives slik (én eller flere elementer kan mangle eller abstraheres):

1. *Stimulus* som er en operasjon som objektet støtter og viser i grensesnittet sitt, evt. med aktuelle argumenter
2. *Triggered events* er hendelser som sendes til andre objekter, f.eks. kall på (meldinger til) andre objekter (*callevent*), signaler som sendes (*signalevent*), eller at objektet skal settes til å vente til noe blir sant (*changeevent*). Hendelser blir kun trigget hvis tilstandsovergangen blir noe av.
3. *Guard condition* er betingelsen for at objektet skal reagere. Det kan anvendes til å vise flere forskjellige tilstandsoverganger på samme stimulus, f.eks. avhengig av om et passord er galt eller korrekt.
4. *Actions*[28] er handlinger som objektet utfører som resultat av hendelsen. I motsetning til *triggered events* er *actions* ting som objektet gjør selv evt. med seg selv. *Actions* blir kun utført hvis tilstandsovergangen blir noe av. Mange ganger kan slike med fordel spesifiseres inne i tilstanden, slik det er gjort her.

Alle konsekvenser av en hendelse kalles samlet for *respons*. Syntaksen fremgår av passordeksemplet nedenfor. Det er også mulig å angi enkelte detaljer inne i tilstandssymbolet, f.eks.



Her sies det at tilstanden ”mottar passord” skal begynne (*entry*) med å sette ekko til passordtegnet ”*”. Mens objektet er i tilstanden (*do*) skal det motta tegn, og når tilstanden avsluttes (*exit*) skal ekko settes normalt igjen og objektet skal sende meldingen *fortsett()* til seg selv. Det er også mulig (men ikke i StarUML) å legge til *include* som viser til et annet tilstandsdiagram som skal inkluderes i dette. Mitt inntrykk er at dette er lite brukt.

Tilstandsoverganger som ikke er angitt, er heller ikke mulige, og objektet skal ikke reagere – evt. *kan* det kaste feil eller gi en tilbakemelding om at det ikke vil gjøre som anmodet. Det er ikke definert noen standard reaksjon på slike feil i UML.

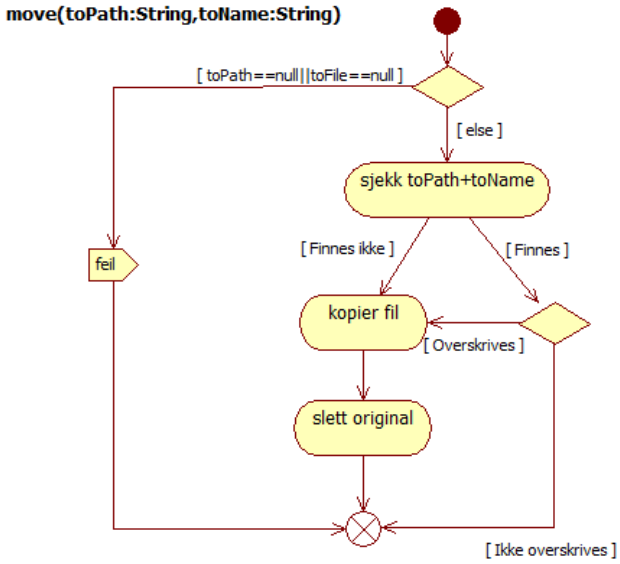
Tilstandsoverganger fra en tilstand til den samme tilstanden, kalles *self-transitions* og er *refleksive*. Når en hendelse ikke fører til hverken trigger, respons eller tilstandsovergang, er det standard at overgangen ikke skal tegnes inn. I fileksemplet er det redundant å tegne inn overgangen som viser at et objekt som er i tilstand "free" fortsetter i samme tilstand hvis det får meldingen *lockfile(free)*.

Aktivitetsdiagram (Activity Diagram)

Slik UML ser det, så er *aktivitetsdiagram* en variant av tilstandsdiagram. Selv synes jeg at det er mer likt et flytdiagram (*flow chart*) som brukes til å beskrive algoritmer. UML tolker imidlertid handlingene som noe som foregår over en viss tid, og følgelig kan de sies å utgjøre *tilstander*. UML kaller dem derfor *action states*. Når en handling er ferdig, går objektet over til å gjøre neste handling, og denne overgangen blir da tolket som en tilstandsovergang – *transition flow*. Hele diagrammet beskriver én operasjon e.l.

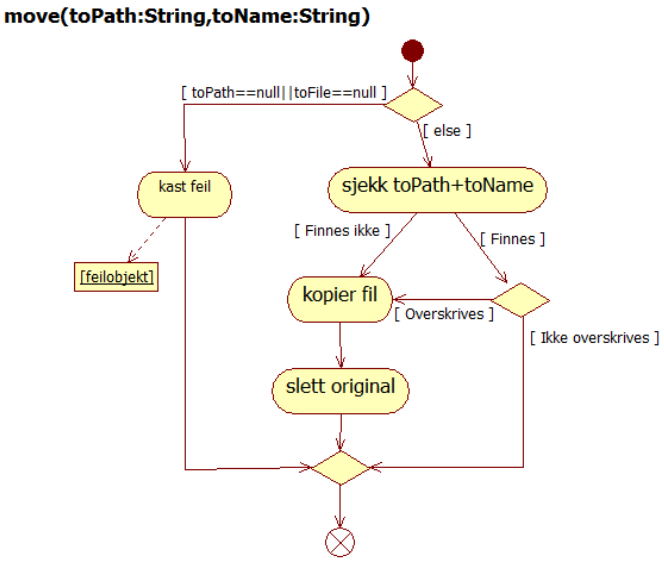
[28] StarUML kaller det *"effects"*.

Her er et eksempel (operasjonen *move()* i File-klassen ovenfor):



De avrundede symbolene beskrives slik: *"An action state is shown as a shape with straight top and bottom and with convex arcs on the two sides"*. En tilstandsovergang trigges ikke av eksterne hendelser (som f.eks. en mottatt melding) men av at forrige aktivitet er ferdig. Aktivitetene kan ha flere utganger (og innganger), avhengig av aktivitetens resultat. Resultatene angis som Boolske uttrykk og kalles *Guard Condition*, da de skal "beskytte" aktiviteten under mot å bli utført hvis ikke uttrykket er sant. Det er også tegnet et rent valg her – de tas på grunnlag av objektets egne attributter (eller som her: Overførte, aktuelle argumenter). Man kan om ønskelig bruke samme symbol for å trekke alternative løp sammen igjen (vist i neste figur).

Hvis kontrollen av parametre viser feil, sendes her et signal "feil". Et slikt signal inneholder jo ingen informasjon om feilen, bare at en feil har skjedd (omtrent som et rødt trafikklys). Hvis man vil sende med mere data om feilen, må man sende et feilobjekt. Det vil se litt annerledes ut:



Aktiviteten ”kast feil” sender her en strøm av objekter kalt en *object flow* (som her bare består av ett objekt). Legg merke til at det da ikke skal tegnes en objektflyt fra aktivitetstilstanden til objektflyten – det er jo ingen tilstandsovergang. Isteden brukes prikket pil.

Flyten kan gå ut i parallelle, synkroniserte løp (tråder), slik det er tegnet i denne tegningen (fra UML-standarden):

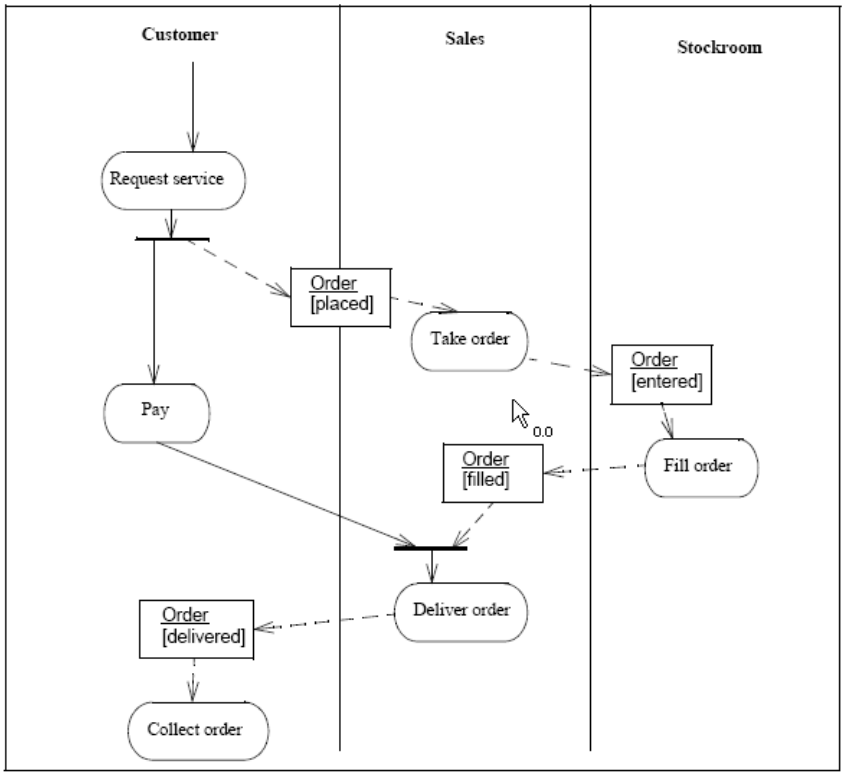


Figure 3-90 Actions and Object Flow

I dette eksemplet går flyten ut i to samtidige løp etter ”Request service”. De kommer sammen igjen før ”Deliver order”. Videre er det vist *Swim lanes*. De brukes bare for visuelt å vise hvem/hvor aktivitetene utføres. Dessuten er objektet ”Order” vist flere ganger, i forskjellige tilstander angitt i parentesen bak navnet.

Det er – som vanlig i UML☺ – *mange* andre notasjonsmuligheter, som jeg ikke går inn på her.

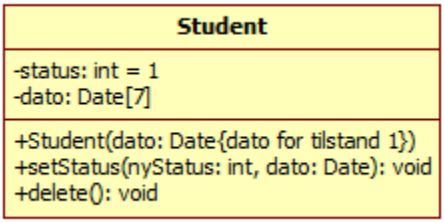
Oppgave til kapittel 6 (tilstands- og aktivitetsdiagram)

På en privat høgskole som selv foretar opptaket (de bruker ikke ”Samordna opptak”), har en student følgende tilstander frem til studenten faktisk er student eller ikke, representert ved en kode (nummeret i listen nedenfor):

1. Søknad om opptak er mottatt. Den sjekkes manuelt av opptakskontoret.
2. Søknaden er mangelfull. Søkeren bes om tilleggsopplysninger. Kan gjentas hvis søkeren gir utilstrekkelige tilleggsopplysninger men opptakskontoret kan også avslå søknaden (til tilstand 3).
3. Søknaden er OK (da er den under behandling av opptakskomiteen)
4. Søknaden er avslått (opptakskomiteen vil ikke ta opp studenten). Det skal skrives ut et avslag ("print avslag").
5. Søkeren fikk plass (søknaden godkjent av opptakskomiteen). Det skal skrives ut et tilbud om plass.
6. Søkeren takket ja til plass. Det skal skrives ut en bekreftelse.
7. Søkeren takket nei til plass. Det skal skrives ut en bekreftelse.

Initialtilstand er tilstand 1. For å bli student må man altså søke med en OK søknad, få plass av opptakskomiteen eller klagenemnden og akseptere plassen.

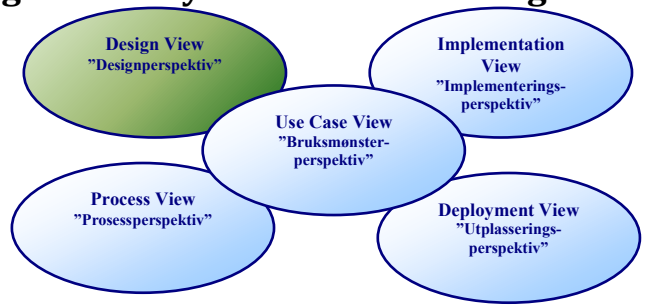
En student som fortsatt ikke er ferdigbehandlet når studieåret begynner, slettes.



- ✓ Attributtet *status* viser gjeldende status som et nummer i henhold til listen ovenfor.
- ✓ Attributtet *dato* er en array med plass til datoer – indeksen referer til tilstandsnummeret – og viser når tilsvarende status inntraff sist. For tilstander som ikke er nådd ennå, er datoen *null*. (Element 0 brukes ikke.)
- ✓ Operasjonen *Student(dato)* er selvsagt konstruktør.
- ✓ Operasjonen *setStatus(nyStatus)* brukes til alle statusendringer. Denne operasjonen kontrollerer inputargumentene – det må være en lovlig tilstandsovergang, og datoen må være rimelig, f.eks. må tilstand 4 inntreffe *samtidig eller etter* tilstand 3. Hvis endringen gikk greit, er attributtet *status* satt lik argumentverdien *nyStatus* og det tilsvarende datoattributt er ajourført. Ellers kastes feil og intet ble endret.
- ✓ Operasjonen *delete()* sletter objektet.

Oppgave: Lag tilstandsdiagram for studenten, og aktivitetsdiagram for *setStatus()*. Bruk StarUML og legg vekt på å få til ryddige diagrammer.

Kapittel 7 – Design View: Dynamisk modellering av samhandling



Ovenfor har jeg omtalt den ene siden av systemets dynamikk, nemlig det enkelte objekts oppførsel, dokumentert med tilstandsdiagram og aktivitetsdiagram. En annen side av dynamikken er den samhandlingen (interaksjonen) som skjer *mellom objekter*.

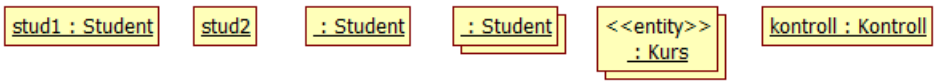
Dokumentasjon av samhandling mellom objekter

Interaksjon (samhandling) mellom objekter vises ved **interaksjonsdiagrammer** (interaction diagrams) som det også er to av. Som tidligere nevnt (forrige kapittel) har de har litt forskjellig fokus. Begge viser hvordan objektene samarbeider gjennom å sende meldinger/signaler til hverandre for å få til en bestemt funksjonalitet. Det gjør de ved å vise enkeltobjekter og hvilke meldinger de sender hverandre (med parametre og svar):

- ✓ **Sekvensdiagrammer** fokuserer på rekkefølgen av meldingene/signalene (derav navnet) over tid. Det fremgår også hvilke objekter som samarbeider, men det er meldingene/signalene og rekkefølgen som er sentral
- ✓ **Samarbeidsdiagrammer** fokuser på hvilke objekter som samarbeider med hvilke. Rekkefølgen av meldingene vises bare gjennom nummerering, så den er vanskeligere å se.

Samarbeidsdiagrammer[29]

I samarbeidsdiagrammer opptrer *objekter*. Det er jo mulig også for klasser å delta i samarbeid med sine klasseoperasjoner, men det er mer sjeldent[30] og ikke alltid implementert i verktøyene. Alle objektene vises med samme symbol som klasser. Man kan se at det er objekter og ikke klasser, fordi navnet er skrevet med ikke-fete typer og er understreket. Det dessuten anbefalt at objektnavn har liten forbokstav.



Fra venstre mot høyre vises her:

- ✓ Objektet *stud1* (objektets navn[31]) som er en instans (objekt) av klassen *Student*.
- ✓ Objektet *stud2* uten angivelse av hvilken klasse den tilhører.
- ✓ Et "anonymt" objekt – det er "ett eller annet objekt" av klassen *Student*.
- ✓ *Flere* anonyme studentobjekter (det er uklart hva det skal bety hvis de er navngitt).
- ✓ Kursobjektene – det er flere av dem – har angitt med stereotype at det er entitetsobjekter (klassen er en entitetsklasse).
- ✓ Kontrollobjektet *kontroll*

Samarbeidet mellom objektene (og evt. klasser) skjer ved at de genererer hendelser for hverandre. Det kan i prinsippet være *signaler*, men vanligst er *meldinger* med/uten returverdi.

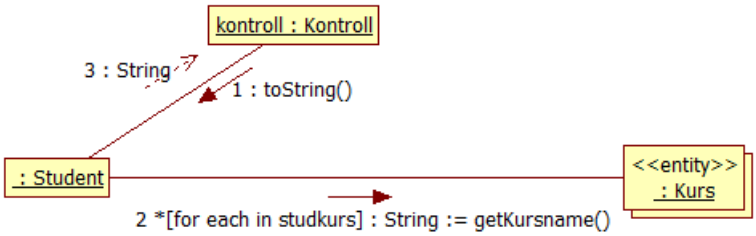
---

[29] Se evt. uml-diagrams.org (http://www.uml-diagrams.org/sequence-diagrams.html) for alle detaljer
[30] Jeg har inntrykk av at det er særlig uvanlig i Java.
[31] Det vil jo være navnet på referansen, altså et attributt eller en variabel.

De fleste meldinger vil være *synkrone* slik at senderen – klientobjektet – vil vente til mottakeren – tjenerobjektet – er ferdig med å behandle meldingen. Ved *asynkrone* meldinger (og signaler som alltid er asynkrone) fortsetter senderen å eksekvere uten å vente på mottakeren – de eksekverer da f.eks. i hver sin tråd, på hver sin maskin e.l. Meldinger som sendes må gjelde en operasjon som mottakeren tilbyr og som er synlig for senderen. Selvsagt må meldingen ha riktig syntaks, altså korrekte argumenter.

Meldinger kan – men må ikke – føre til svar fra mottakeren. Når meldingen gjelder en funksjon og meldingen er synkron, antar man vanligvis at det gis svar og det tegnes da ikke spesielt inn. Hvis en asynkron melding gir svar, er det vanlig å tegne inn svaret som en egen hendelse.

Meldinger kan også skape og slette objekter, men det tar jeg ikke opp her.

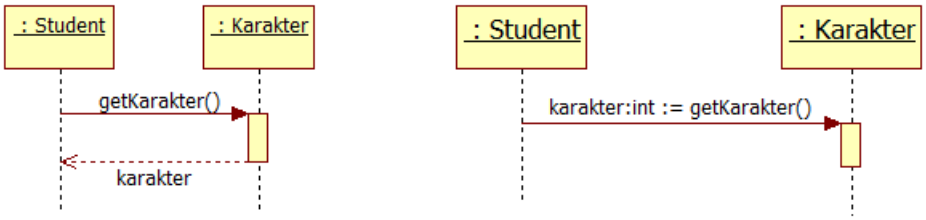


Her vises tre objekter som samarbeider. Handlingene starter med at *kontroll* sender en synkron melding *toString()* til et *Student*-objekt. *Student*-objektet sender da meldingen *getKursnavn()* til alle kursene i sin *studentkurs*-samling og hver av disse meldingene gir en streng i retur. Til slutt returnerer *Student*-objektet en streng til *kontroll*. Meldingene er sekvensielt nummerert. Iterasjonen er angitt som en "guard condition" med tillegg av stjerne. Her kan det også stå vilkår for når meldingen skal sendes, som et Boolsk uttrykk f.eks. *[x>y]:bytt(x,y)*.

Sekvensdiagrammer

I sekvensdiagrammer vises også objekter og meldinger. Notasjonen er nesten den samme som for samarbeidsdiagrammer (og takk for det☺!).

I sekvensdiagrammer tydeliggjøres rekkefølgen og tidsaspektet og derfor er det behov for å vise når objektene er aktive og når de er passive. Det gjøres gjennom en livslinje ("lifeline") loddrett under objektet. Livslinjen er prikket når objektet venter, og et stående rektangel når objektet gjør noe. Meldingene vises fra livslinje til livslinje og rekkefølgen er alltid ovenfra og nedover. En samling meldinger kan grupperes i en ramme ("frame") og navngis og deles i flere forløp. Det går jeg ikke nærmere inn på her.

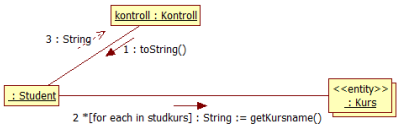


Her ser man til venstre at et anonymt *Student*-objekt sender den synkrone meldingen *getKarakter* til et anonymt *Karakter*-objekt. Etter å ha håndtert meldingen en tid (det angis ikke hvor lang tid – det vil jo også avhenge av maskinhastighet, belastning og mye annet) svarer *Karakter*-objektet med en karakter. Jeg har spesielt valgt å ta bort nummereringen av

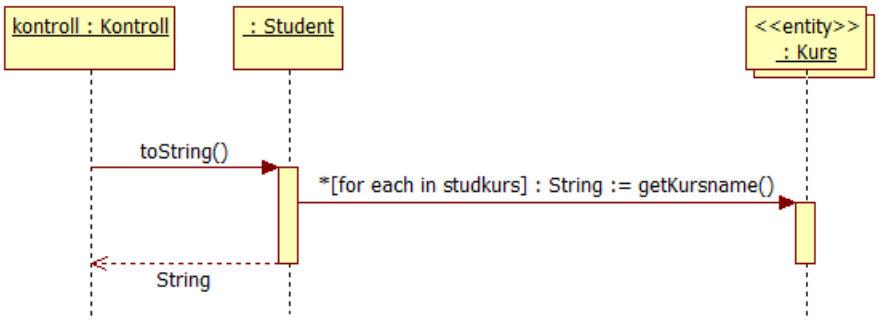
meldingene, da den er implisitt ovenfra og nedover. Til høyre ser man det samme, også med en synkron melding men med returen angitt direkte. Der her jeg også angitt returtypen. Ofte angis *bare* returtypen, da det jo egentlig er klienten som bestemmer hvor returverdien skal tas vare på. Poenget her er at det er *Karakter*-objektets attributt *karakter* som returneres, men det er også rimelig opplagt pga navnet *getKarakter*..

Sammenheng mellom sekvens- og samarbeidsdiagrammer

Sekvens- og samarbeidsdiagrammer viser nøyaktig det samme, bare med forskjellig fokus. Derfor kan edb-verktøy automatisk generere den ene på grunnlag av den andre. Her tar jeg samarbeidsdiagrammet vist ovenfor (gjentatt):



Jeg ber nå StarUML om automatisk å generere sekvensdiagrammet som tilsvarer dette samarbeidsdiagrammet. Det fikser jeg litt på[32] og da blir det slik:

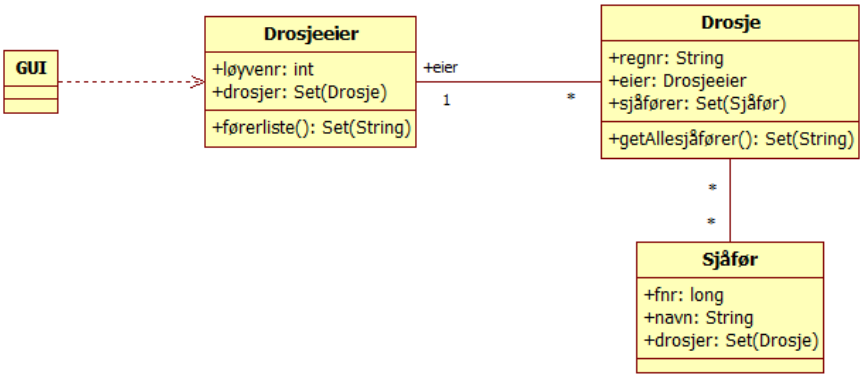


Dette viser altså nøyaktig det samme som samarbeidsdiagrammet.

Eksempel: Drosjer med flere alternativer

Forenklet klassediagram

Legg merke til at det er laget to operasjoner som ikke er standard tilgangsoperasjoner o.l.

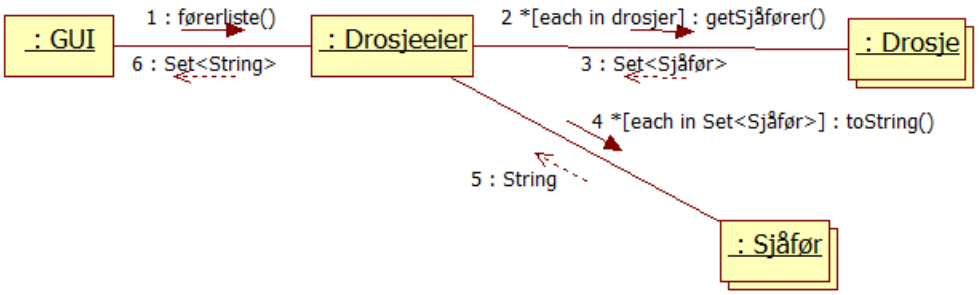


Nedenfor skriver jeg om objektene som om de faktisk var drosjeeiere, drosjer osv. Det er selvsagt upresist, men det gjør det ofte lettere å følge tankegangen, synes jeg.

[32] Rekkefølge, lengde på de aktive boksene o.l. Jeg fjerner også sekvensnumrene, fordi jeg mener rekkefølgen her kommer tydelig frem uten dem.

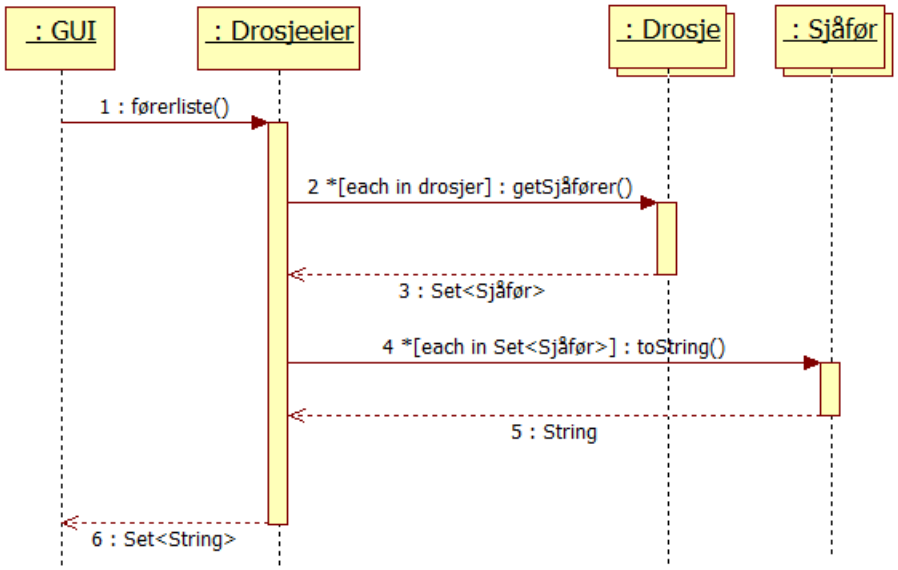
Samarbeidsdiagram versjon 1

En drosjeeier blir bedt om lage en liste over alle sjåførene som kjører hans biler. Drosjeeieren ber alle drosjene sine – mengden *drosjer:Set<Drosje>* – om å oppgi referanse til alle sine sjåfører. Drosjeeieren har nå en komplett samling med referanser til sine sjåfører (uten dubletter hvis de returnerte mengdene kombineres med union) og kan be hver og en av dem om å returnere sitt nr og navn som streng. Disse strengene samles i en mengde som returneres GUI-objektet.



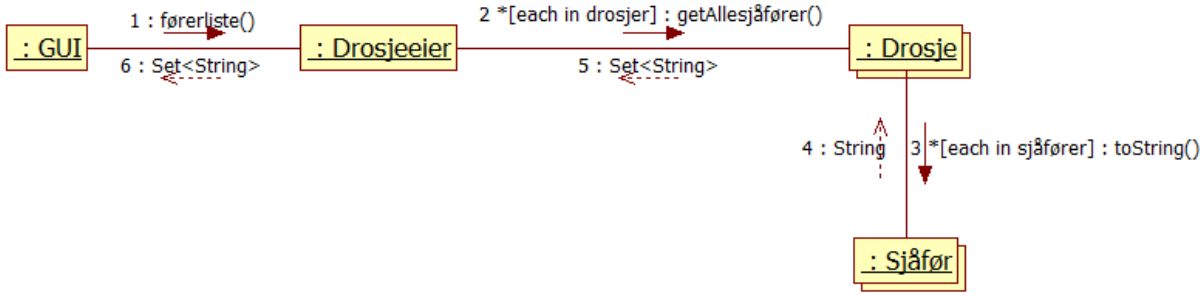
Sekvensdiagram versjon 1

Her er samarbeidsdiagrammet gjort om til sekvensdiagram av StarUML. Dette diagrammet viser det samme som samarbeidsdiagrammet, men det er lagt mer vekt på å vise *rekkefølgen* (sekvensen) av meldingene.

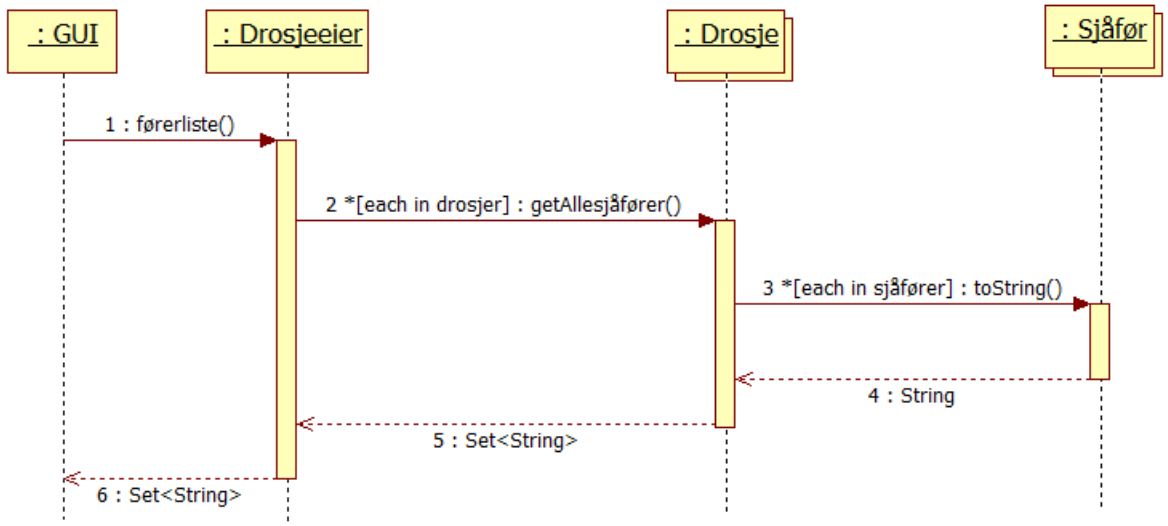


Samarbeidsdiagram versjon 2
Utfører samme funksjon, men på en annen måte enn diagram 1.

En drosjeeier blir bedt om å lage en liste over alle sjåførene som kjører hans biler. Drosjeeieren ber alle sine drosjer – mengden *drosjer:Set<Drosje>* – om å returnere nr og navn på alle deres sjåfører som en mengde strenger. Hver drosje må da be alle sine sjåfører i mengden *sjåfører:Set<Sjåfør>* om å oppgi nr og navn som streng. Disse samles til en mengde og returneres drosjeeieren. Når drosjeeieren har fått svar fra alle drosjene, kan han returnere hele mengden til GUI-objektet (uten dubletter ved å samle dem med union i én mengde).

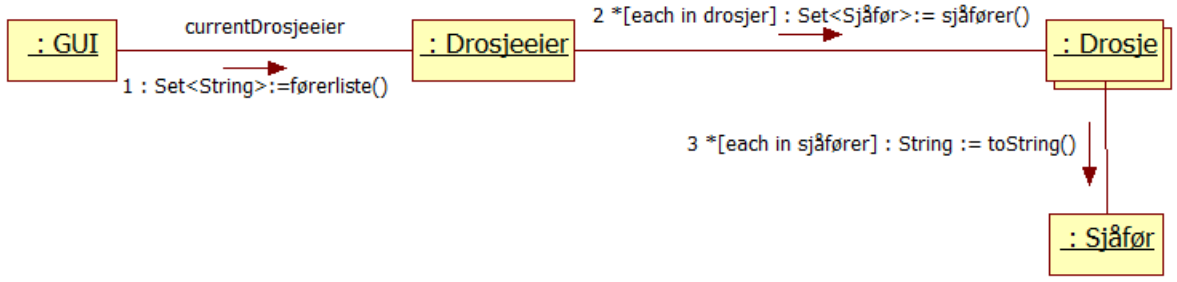


Sekvensdiagram versjon 2
Viser det samme som samarbeidsdiagram versjon 2.



Alternativ syntaks for meldinger

Her tegnes ikke returen – den er implisitt i meldingen. Jeg har også gitt assosiasjonen GUI⇔Drosjeeier et navn (GUI-objektets navn på referansen).



Jeg synes rekkefølgen kommer dårligere frem her – det kan se ut som melding 1 returnerer *Set(String)* før melding 2 har returnert og tilsvarende for melding 2 og 3. I virkeligheten må jo melding 2 og 3 gjennomføres og returnere verdi *før* melding 1 kan få svar.

Signaler og aktive klasser

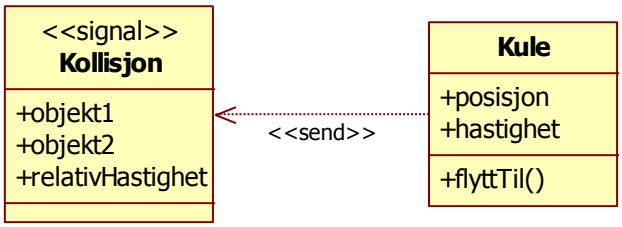
Signaler

Meldinger har jeg skrevet mye om ovenfor – de representerer *kall på objektenes operasjoner*. De sendes til et objekt og har et navn (som tilsvarer objektets operasjonsnavn) og det kan vedlegges data i form av aktuelle argumenter. De er en anmodning om at objektet skal gjøre noe, og kanskje returnere et svar og/eller programkontrollen. De kan være asynkrone (til en annen tråd) men er vanligvis synkrone.

Et objektorientert system kan også sende og motta *signaler*. Et signal er en beskjed om at noe har hendt, og vil avbryte vanlig eksekvering av det objektet som mottar det (det er en hendelse som skal håndteres). Signaler kan inneholde data om hendelsen men de er begrensede. Signaler er asynkrone og kan ikke besvares (dog kan et objekt som har mottatt et signal, selv generere et nytt signal, slik en exception i Java kan videresendes ”oppover” i systemet til det kommer til et objekt som håndterer det). Ordet brukes i mange sammenhenger, f.eks. ”kjøpssignal” på børsen, ”nødsignal” til sjøs, ”exception” i Java, ”ringesignal” på mobilen, ”nervesignal” i biologien, ”gulblink” i et trafikklys osv.

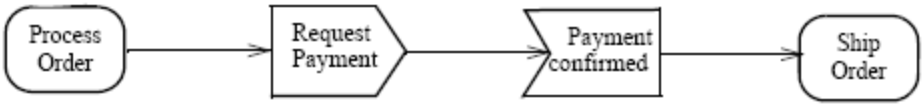
Signaler kan sendes adressert til et annet objekt, eller ”kringkastes” (eng. *broadcast*) til mange. I siste tilfelle er det opp til de som mottar signalet om de mener at det berører dem. Signaler er ikke avhengig av synlighet (slik operasjoner er).

Signaler kan evt. modelleres som en klasse, med stereotypen <<signal>>. Her er det brukt i en modell av billiardspill der kulene er objekter som kolliderer:



Hvis en kule oppdager kollisjon med en annen kule, sendes et signal om kollisjon med angivelse av objektene som er involvert og den relative hastigheten dem imellom. Tanken er da at et annet objekt skal håndtere denne kollisjonen.

Signalene modelleres også i aktivitetsdiagrammer (fra UML 2.0 standarden):

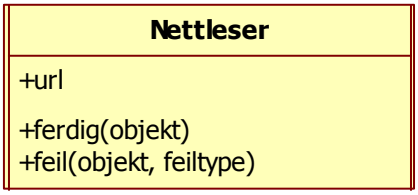


"Request Payment" er et *sendt* signal, mens "Payment confirmed" er et *mottatt* signal.

Aktive klasser

I realtime (RT) systemer vil det oppstå mange prosesser og tråder som eksekverer parallelt – på samme eller på flere maskiner (noder). Denne samtidigheten må styres, og det gjøres med signaler som håndteres av objekter av en *aktiv klasse*.

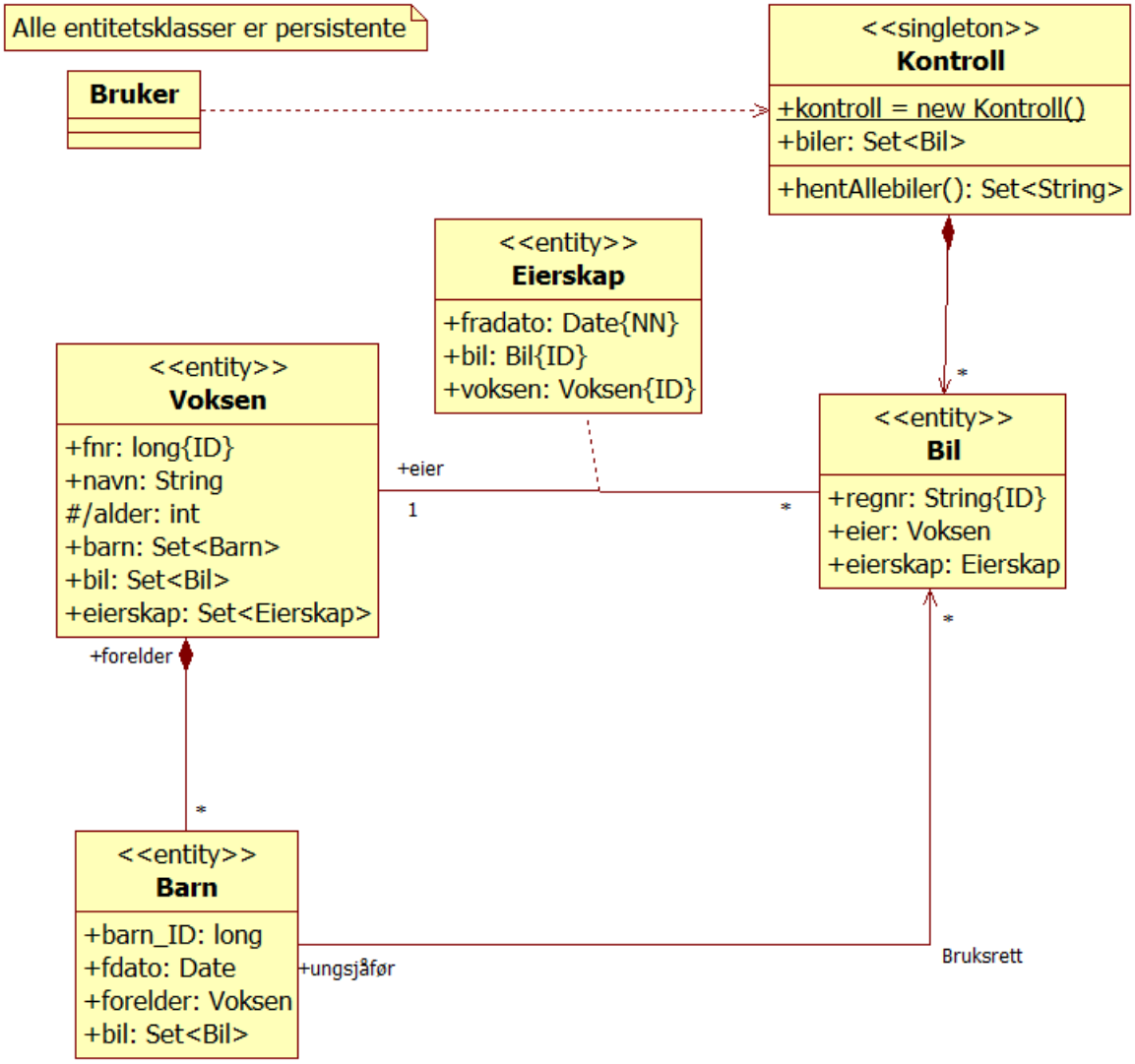
Til forskjell fra andre klasser, vil objekter av en aktiv klasse styre seg selv og ikke bli styrt av meldinger de mottar. For det andre vil de kjøre uavhengig av de andre, og gjerne så lenge applikasjonen kjøres. Aktive klasser håndterer isteden signaler og sender selv signaler (”start”, ”stopp”, ”vent” osv.) til andre objekter. Aktive klasser tegnes med en ekstra strek loddrett på hver side:



Den aktive klassen instansieres til aktive objekter som ”eier” prosesser og tråder.

Oppgave til kapittel 7 (samarbeids- og sekvensdiagram)

Gitt følgende klassediagram:



Vi har sett denne modellen i en tidligere oppgave[33] – men her er også alle assosiasjonene realisert med attributter. Vi forutsetter at det skal legges til operasjoner etter standarden

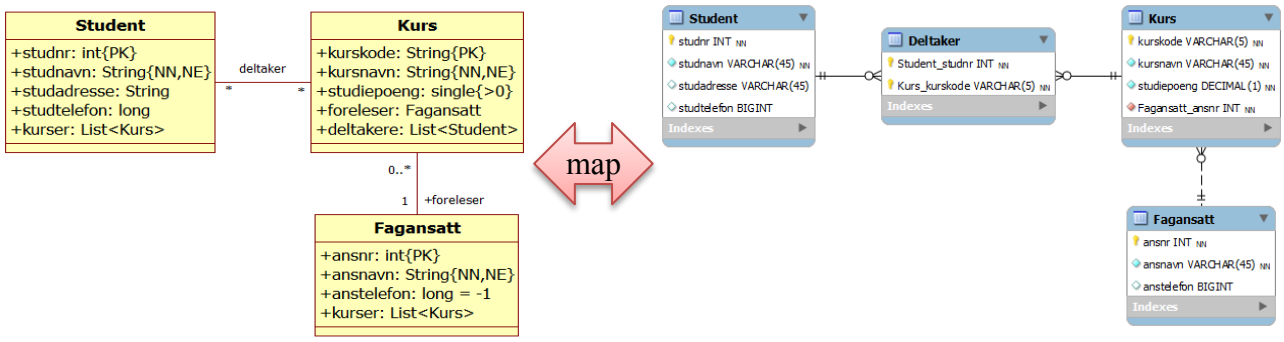
Det er vist én, ekstra metode *Kontroll.kontroll.hentAllebiler()*. Hver streng i mengden som denne operasjonen returneres, inneholder alle opplysninger om bilene med tilhørende eier og alle barn som kan kjøre bilen. Det er ikke nødvendig å ta med data fra *Eierskap*-objektene. Man ønsker maksimal delegering, slik at *Kontroll.kontroll* ber *Bil*-objektene om å returnere seg selv, eier og barn osv. Innfør de operasjonene du mener klassene trenger for å få dette til.

Oppgave: Vis med samarbeids- og sekvensdiagram hvordan man kan realisere en brukerfunksjon som returnerer en mengde strenger med kallet *Kontroll.kontroll.hentAllebiler():Set<String>*.

[33] Oppgave til kapittel 5 (standardmetoder)

Kapittel 8 – Persistens av objekter: OOP & RDBMS

Note: Dette kapitlet er ikke direkte relatert til UML-standarden men stoffet her er viktig for OOP-programmere.



Illustrasjon: *Til venstre en modell av et objektorientert system, til høyre en modell av en relasjonsdatabase. Siden de to teknologiene har helt forskjellige paradigmer, er det store forskjeller i modellene – både i detaljer (datatyper, referanser o.a.) og overordnet (antall tabeller vs klasser, indekser o.a.). "Map" er en dokumentasjon av hvordan de to modellene samsvarer.*

Innledning

Når man deklarerer en kunde *Private Kunde kunde = new Kunde();* så legges det nye objektet i RAM. ”Scope” er begrenset til referansens (variabelens) scope og levetiden til sesjonen. Objektet blir borte når sesjonen avsluttes, strømmen går o.l. Objektet er da *transient*. For mange objekter er det OK, fordi objektet er flyktig i seg selv, f.eks. grenseobjekter som håndterer påloggede brukere og printerobjekter og kontrollobjekter som håndterer køer, samtidighet osv.

For objekter som skal ta vare på data – typisk (men ikke bare) entitetsobjekter, holder det ikke. Da må man ha *persistente* objekter, som lagres på et ytre lagringsmedium mellom sesjoner. Da kan man bruke:

1) ”Flate” filer (direkte eller sekvensielle)
2) Nettverksdatabaser og hierarkiske databaser
3) RDBMS (relasjonelle databaser) og ORDBMS (objektrelasjonelle databaser)
4) NOSQL-databaser, f.eks.
   a) Dokumentbaserte databaser
   b) OODBMS (objektorienterte databaser)
   c) Prevayler som logger alle endringer og kjører loggen ved neste sesjonsstart

Alternativ 1 og 2 blir ”gammeldags” og tungvint, f.eks. må man selv håndtere kontroll med entitetsintegritet, referanseintegritet og samtidighet (med låsing, transaksjoner osv.). De ser jeg følgelig bort fra her. Alternativ 4 er pensum i et annet kurs. Her skal jeg diskutere alternativ 3 med lagring av objekter i relasjonsdatabaser og objektrelasjonelle databaser, som er de mest brukte. Kombinasjonen OOP og RDBMS må derfor antas å være svært vanlig.

De objektrelasjonelle databasesystemene – som f.eks. Oracle – har en viss objektliknende funksjonalitet. I praksis bidrar det ikke ved lagring av objekter, og følgelig behandler jeg ikke det her. Da blir det likt, enten man lagrer i en relasjonell eller en objektrelasjonell database.

Objekter og RDBMS

Det er ganske vanlig å bruke OOP men lagre persistente objekter i en RDBMS. *Hvor* vanlig det er, er vanskelig å si, men de aller fleste virksomheter – antakelig over 90% bruker RDBMS og ORDBMS til lagring av data. ("Flate filer" brukes nesten bare til logg, ini-filer og filer knyttet til bare én applikasjon.) Samtidig er bruken av OOP stigende – flertallet av de programspråkene som er mest i bruk er objektorienterte.

Når man bruker RDMBS sammen med objektorienterte programmer får man problemer med å opprettholde OOPs hovedprinsipper

- ✓ innkapsling i begge betydninger:
  - abstraksjon = skjult implementering f.eks. ukjent programkode for metodene og private data/metoder
  - data/metoder holdes samlet
- ✓ arv
- ✓ polymorfisme
- ✓ ”containere”, f.eks. set, bag, list, array osv.
- ✓ referanser, dvs. direkte pekere til objekter

fordi disse mekanismene ikke finnes i RDBMS. I tillegg må RDBMS ha *metadata* (*schema*) som deklarerer tabeller, data, brukere, indekser osv.

RDBMS har *kun* tabeller (matematisk kalt *relasjoner*) og må ha minst ett felt, kalt primærnøkkel, som *skal* ha unike verdier. I OOP regnes to objekter som forskjellige, bare de er lagret hvert sitt sted i RAM[34].

I OOP kan et objekt være inkludert (som ”attributt”) i et annet objekt. I RDBMS er det *ikke* tillatt med en tabell inne i en annen tabell[35].

Hvis man skriver alle våre persistente objektklasser

- ✓ helt uten arv
- ✓ med unike nøkkelfelt
- ✓ med bare public attributter
- ✓ helt uten polymorfisme (følger av første punkt)
- ✓ med bare enkle datatyper

da kan klassestrukturen lett overføres til en RDBMS, men OOP blir det ikke! I praksis vil man ikke bindes så sterkt, og man får et problem med å lagre de persistente objektklassene til datatabeller i RDBMS.

Mapping av OOP til RDBMS

Med mapping forstår man en omgjøring av strukturen. Hensikten er å lage regler for overgang fra en struktur – her OOP – til en annen – her RDBMS. Mappingen viser hvordan strukturen i den ene henger sammen med strukturen i den andre.

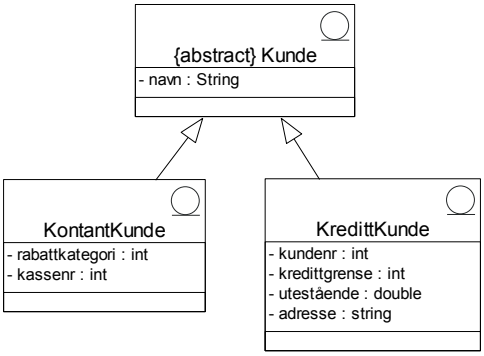
[34] I *OODBMS* regnes de ofte forskjellige bare de har minst én attributtverdi forskjellig, men det kan *variere* hvilket attributt det er. Slike databaser kan ofte også selv, automatisk legge til et eget ID-attributt, kalt Object Identifier, forkortet OID.

[35] I Oracle *er* det faktisk tillatt, men det har så mange ulemper at det er lite brukt.

Det største problemet oppstår med arv og med attributter som er samlinger. For å løse problemene, kan man bruke fire forskjellige strategier for mapping:

- ✓ Vertikal mapping = Alle klasser får sin egen tabell.
- ✓ Horisontal mapping = Bare konkrete klasser mappes, alle arvede attributter tas med der.
- ✓ Filtrert mapping = Bare øverste klasse mappes, alle underliggende attributter trekkes opp dit og det legges til en "type"-attributt som angir hvilken klasse en post hører til.
- ✓ Generisk mapping = En fast mapping uansett modell i OOP.

Når det skal benyttes en relasjonsdatabase, må det stilles krav til klassemodellen. Alle klasser må ha et eksplisitt identifikatorattributt (det holder ikke med adressen/referansen til objektet slik det gjør i objektorientering da det er en flyktig identifikator), de må ha atomære attributter (attributter som er strukturer som vektorer, arrays, andre klasser og liknende strukturer må løses opp) og alle relasjoner må realiseres med fremmedattributter (da følger det også at alle mange-til-mange assosiasjoner, og assosiative klasser, må objekteres).
Nedenfor er et eksempel, basert på følgende OOP-modell for de persistente klassene:

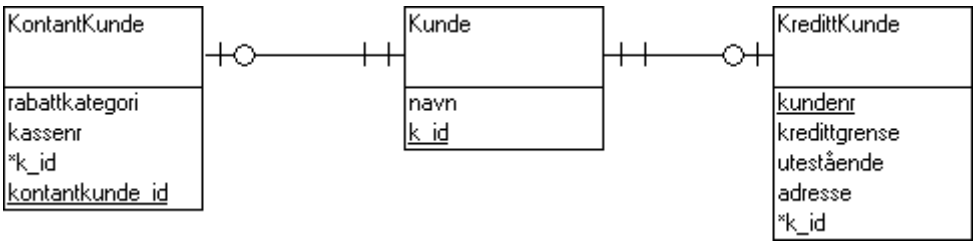


A) Vertikal mapping (separasjon)

Alle klasser mappes til sin egen tabell. Man innfører primærnøkkel (PK) etter behov. Det legges til en logisk peker fra subklasse til metaklasse.

```
Kunde (navn, k_id)
KontantKunde (rabattkategori, kassenr, *k_id, kontantkunde_id)
KreditKunde (kundenr, kredittgrense, utestående, adresse, *k_id)
```

Her er Kontantkunde.kontantkunde_id unødvendig, da k_id også kan brukes som PK.



B) Horisontal mapping (partisjonering)

Bare de *konkrete* klassene mappes med alle attributter – også de som er arvet – til hver sin tabell. Innfører PK etter behov.

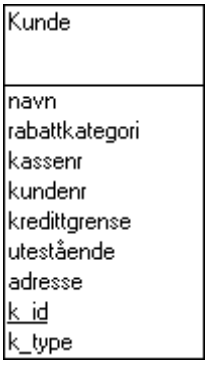
```
KontantKunde (navn, rabattkategori, kassenr, kontantkunde_id)
KredittKunde (navn, kundenr, kredittgrense, utestående, adresse)
```

C) Filtrert mapping (absorpsjon)

Alle klassene i et hierarki slås sammen til én tabell. I tillegg kreves da en attributt som angir hvilken klasse posten tilhører, og selvsagt en PK.

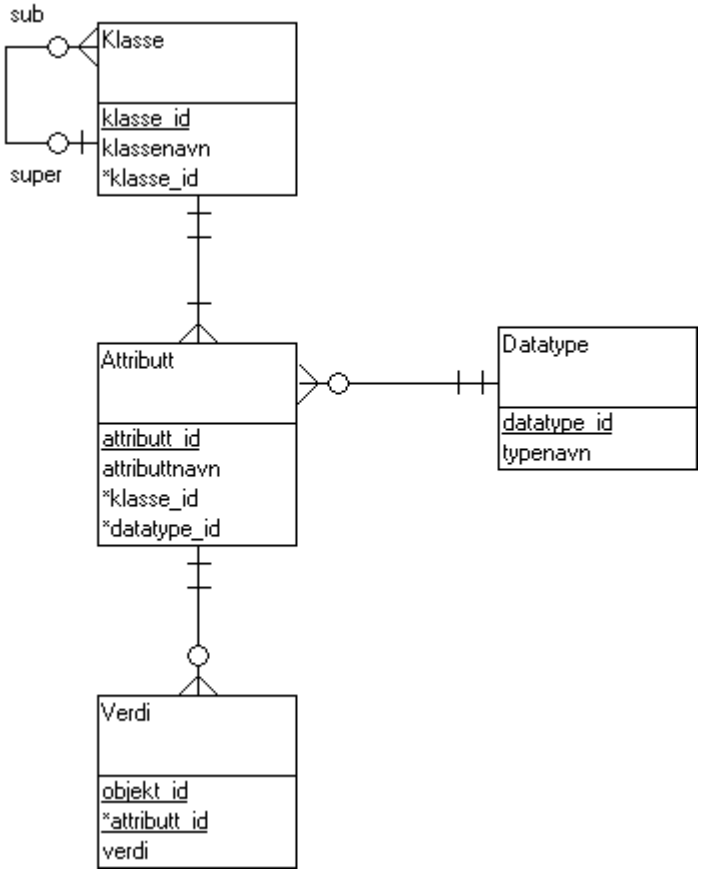
```
Kunde (navn, rabattkategori, kassenr, kundenr, kredittgrense, utestående,
adresse, k-id, k_type)
```

Her kunne det være fristende å bruke *kundenr* som PK, men det går ikke fordi kontantkunder ikke har kundenr.



D) Generisk mapping

Det brukes en *fast* mapping med tabellene *Klasse*, *Attributt*, *Datatype* og *Verdi* med relasjoner dem imellom (i tillegg kan egenrelasjonen til Klasse entitetiseres). Tabellen inneholder mest metadata – verdiene lagres bare i tabellen *Verdi*. Denne er både vanskelig å bruke og å forstå, og diskuteres ikke nærmere her.

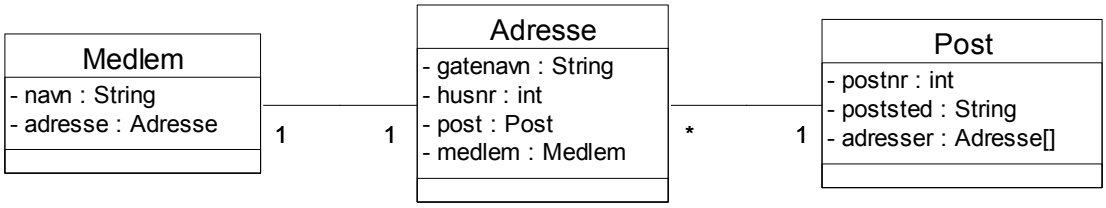


Mapping av noen spesielle situasjoner

Eksemplet ovenfor var svært enkelt fordi

- ✓ det var ingen assosiasjoner, hverken en-til-mange eller mange-til-mange
- ✓ det var bare ett nivå med arv, kunne vært flere
- ✓ det var ingen objekter eller objektstrukturer (ADTer) blant attributtene

Det er eksempler på de to første forenklingene i den eksamensoppgaven som er gjengitt nedenfor, men ikke den tredje forenklingen. Her er et eksempel med en objektstruktur som attributt:



Her tenker man seg først litt om. Man ser at *Adresse* refererer til et *Post*-objekt, som på sin side refererer til mange *Adresse*-objekter. Det er et resultat av assosiasjonen mellom dem. I relasjonsdatabasen bruker man bare den ene (”fra mange til én) – den andre kan man altså se bort fra. Videre ser man at det er en én-til-én assosiasjon mellom *Medlem* og *Adresse*. Da har man følgende alternativer:

1) Slå sammen objektene til én tabell i databasen og ikke lage noen fremmednøkkel. Behovet for fremmednøkler bortfaller.
2) Lage to tabeller, men legge fremmednøkkel i bare den ene av dem
3) Lage to tabeller, men legge fremmednøkkel i begge (det vil være uvanlig i relasjonsdatabaser fordi det er redundant).

Løsning 1:

```
Medlem (navn, gatenavn, husnr, *postnr, medlem_id)
Post (postnr, poststed)
```

Løsning 2:

```
Medlem (navn, *adresse_id, medlem_id)
Adresse (gatenavn, husnr, *postnr, adresse_id)
Post (postnr, poststed)
```

Løsning 3:

```
Medlem (navn, *adresse_id, medlem_id)
Adresse (gatenavn, husnr, *postnr, *medlem_id, adresse_id)
Post (postnr, poststed)
```

Generelle betraktninger

Noen attributter er det *ikke nødvendig å lagre*, f.eks.

- ✓ attributt som er der av praktiske grunner, men som kan finnes/beregnes ved behov (f.eks. antall noder i en liste)
- ✓ attributt som har verdi bare midlertidig (f.eks. *isDirty* som angir om noe er endret)
- ✓ attributt som bare har mening i OOP (f.eks. *nesteKunde* – en peker til neste objekt i en liste)

Noen attributter *må legges til* i RDBMS, f.eks.

- ✓ En identifikator, i form av en verdi, kreves i RDBMS
- ✓ Logisk peker til metaklassen i en vertikal mapping, eller en type-identifikator i en filtrert mapping

Noen *strukturer må gjøres om helt*, f.eks.

- ✓ attributt som utgjør en struktur (en ADT/samling som attributt)
- ✓ attributt som er et objekt
- ✓ mange-til-mange relasjoner
- ✓ multippel arv, der det støttes

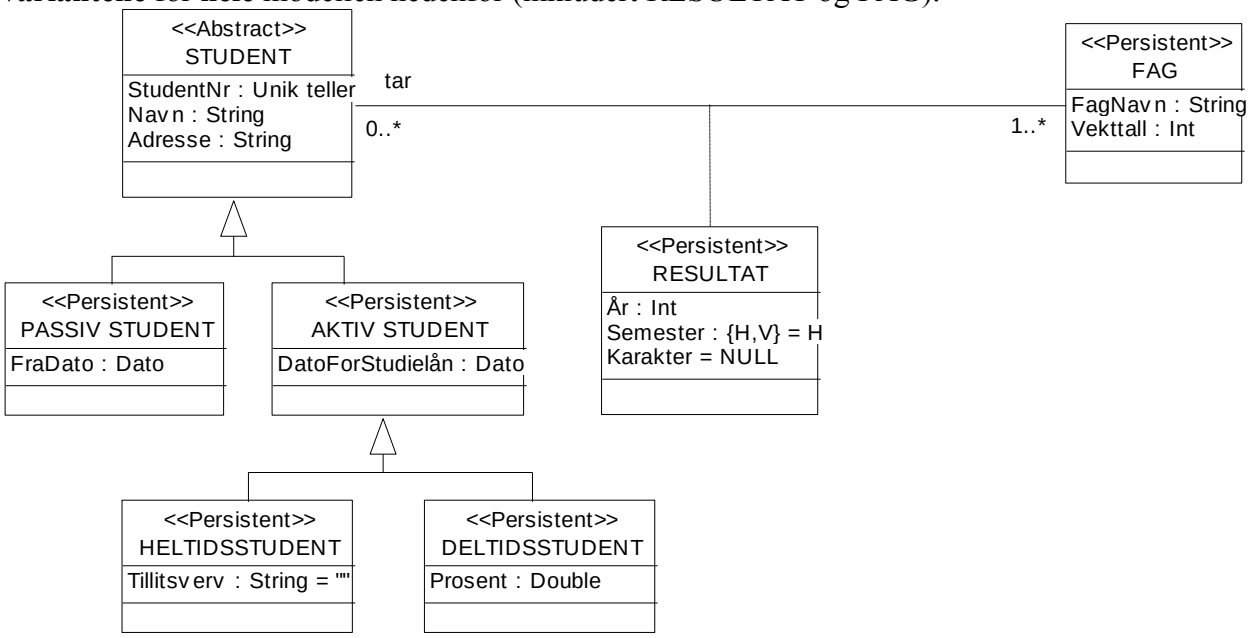
Aggregeringer (”består av”) krever triggere og/eller kaskadeeffekter, siden delene må slettes når aggregeringen slettes.

RDBMS er like forskjellig fra OOP som f.eks. printere, bruker osv., og det anbefales av mange at de håndteres av grenseklasser som finner, henter/bygger objekter, lagrer, sletter og endrer databasen i takt med objektene i RAM. (Selv har jeg funnet at det kan være enklere å inkludere håndtering av databasen direkte i en kombinert grense- og kontrollklasse – se omtalen av DDD på slutten av kapittel 5.)

Det finnes noe programvare som er i stand til å mappe et OOP automatisk til ønsket database etter en angitt strategi.

Eksempel på mapping
Fra eksamen høsten 2001

Oppgave
Forklar meget kort prinsippene for de tre måtene å "mappe" på, og gjennomfør så **alle de tre variantene** for **hele** modellen nedenfor (inkludert RESULTAT og FAG).



Fra sensorveiledningen (løsningsforslag)
Primærnøkler er understreket, fremmednøkler merket med *. Det kan synes rart, men oppgaven synes å tyde på at det finnes studenter som er aktive, uten å være hverken heltidsstudent eller deltidsstudent (AKTIV STUDENT er persistent). Oppgaven er løst slik nedenfor.

Vertikalt – alle klassene mappes til en egen tabell

```
FAG (FagNavn, Vekttall, FAG-ID)
RESULTAT(År, Semester, Karakter, *FAG-ID, *StudentNr)

STUDENT(StudentNr, Navn, Adresse)
PASSIV-STUDENT(FraDato, PASSIV-STUDENT-ID, *StudentNr)
AKTIV-STUDENT(DatoForStudielån, AKTIV-STUDENT-ID, *StudentNr)
HELTIDSSTUDENT(Tillitsverv, ~~DatoForStudielån~~, HELTIDSSTUDENT-ID,
*StudentNr)
DELTIDSSTUDENT(Prosent, ~~DatoForStudielån~~, DELTIDSSTUDENT-ID, *StudentNr)

Alternativt, med StudentNr som ID:
PASSIV-STUDENT(FraDato, *StudentNr)
AKTIV-STUDENT(DatoForStudielån, *StudentNr)
HELTIDSSTUDENT(Tillitsverv, ~~DatoForStudielån~~, *StudentNr)
DELTIDSSTUDENT(Prosent, ~~DatoForStudielån~~, *StudentNr)
```

- ✓ *StudentNr* er unik og kan brukes som ID. De som har *StudentNr* som fremmednøkkel, kan også bruke den som ID.
- ✓ Det kan være fristende å bruke *FagNavn* som ID i FAG. Det synes jeg er tvilsomt.
- ✓ HELTIDSSTUDENT og DELTIDSSTUDENT: Disse må arve attributtene fra AKTIV STUDENT fordi den er persistent.

Horisontalt – bare de konkrete klassene mappes

```
FAG (FagNavn, Vekttall, FAG-ID)
RESULTAT(År, Semester, Karakter, *FAG-ID, *StudentNr)
PASSIV-STUDENT(StudentNr, Navn, Adresse, FraDato)
AKTIV-STUDENT(StudentNr, Navn, Adresse, DatoForStudielån)
HELTIDSSTUDENT(StudentNr, Navn, Adresse, Tillitsverv, DatoForStudielån)
DELTIDSSTUDENT(StudentNr, Navn, Adresse, Prosent, DatoForStudielån)
```

- ✓ Her har jeg gjennomført brukt *StudentNr* som ID (den arves av alle de andre klassene). Hvis man ikke gjør det, vil man få store problemer med relasjonen RESULTAT-STUDENT når sistnevnte klasse splittes i fire tabeller.

Filtrert – alle klassene i et hierarki slås sammen til én tabell

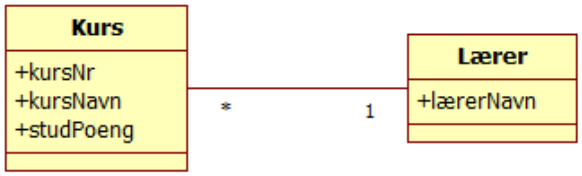
```
FAG (FagNavn, Vekttall, FAG-ID)
RESULTAT(År, Semester, Karakter, *FAG-ID, *StudentNr)
STUDENT(StudentNr, Navn, Adresse, FraDato, DatoForStudielån, Tillitsverv,
Prosent, Klasse)
```

- ✓ En vanlig feil er å mangle attributtet *Klasse* eller tilsvarende i STUDENT.

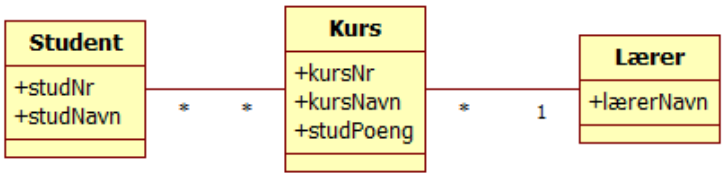
Oppgave til kapittel 8 (mapping)

Nedenfor finner du mange klassediagrammer merket A, B osv. Lag en mapping til RDBMS for hver av dem, med alle de tre vanlige strategiene (der hvor det blir forskjell).

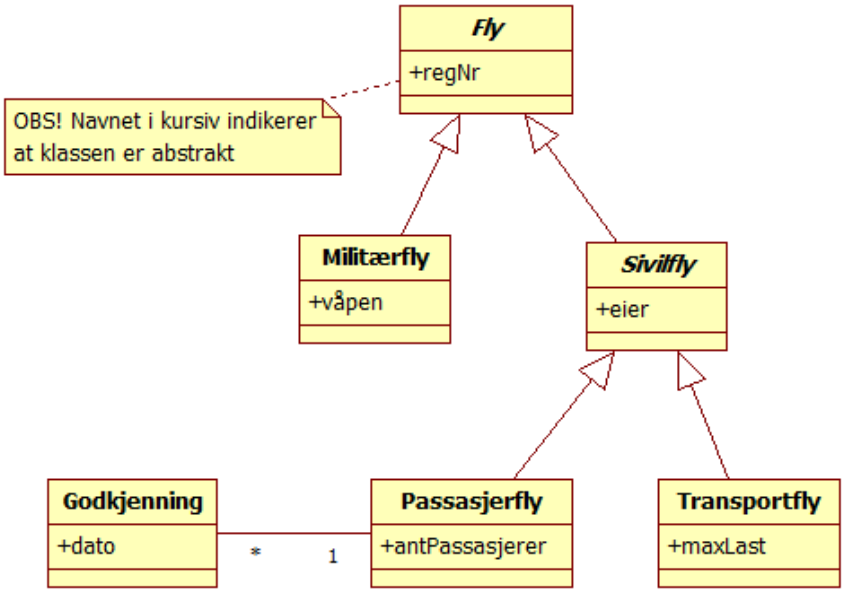
A



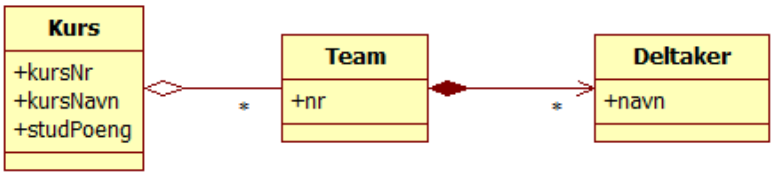
B



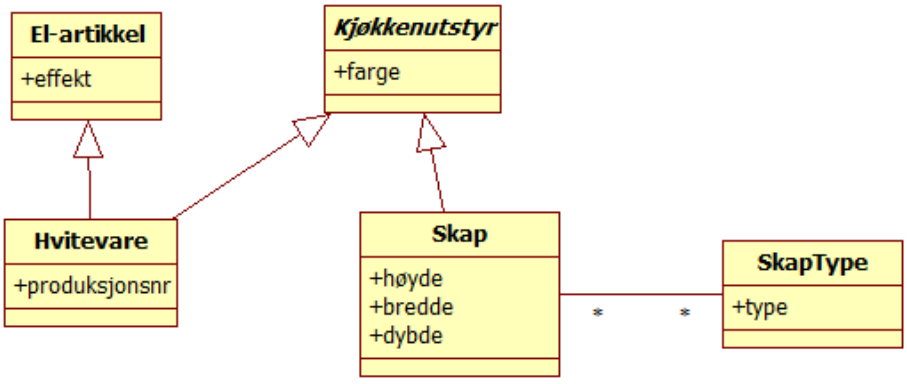
C



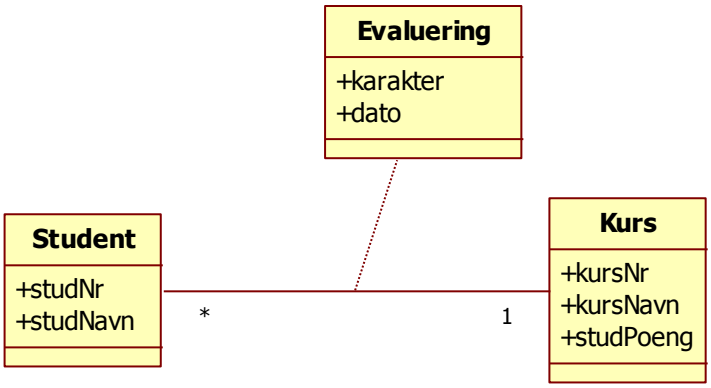
D



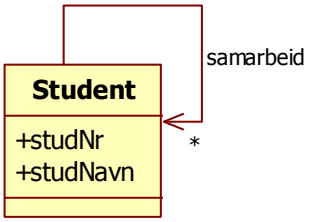
E



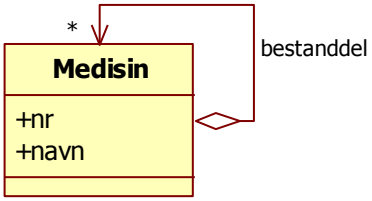
F


Evaluering
+karakter
+dato
Student
+studNr
+studNavn
Kurs
+kursNr
+kursNavn
+studPoeng
*
1

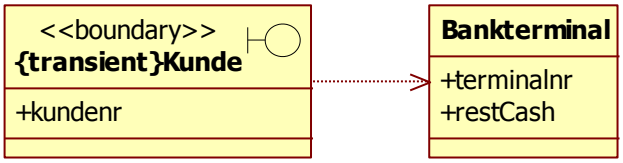
G


samarbeid
Student
+studNr
+studNavn
*

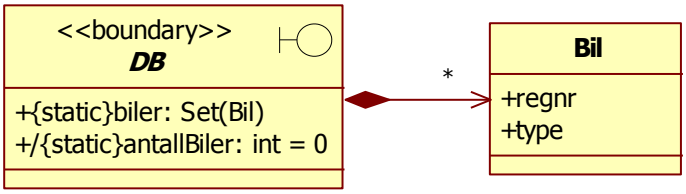
H


*
bestanddel
Medisin
+nr
+navn

I

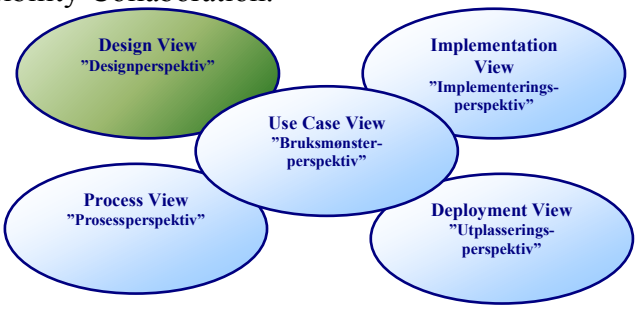

<<boundary>>
{transient}Kunde
+kundenr
Bankterminal
+terminalnr
+restCash

J


<<boundary>>
DB
+{static}biler: Set(Bil)
+/{static}antallBiler: int = 0
*
Bil
+regnr
+type

Kapittel 9 – Design View: CRC cards

CRC = Class-Responsibility-Collaboration.



Bakgrunn

CRC-kort ble opprinnelig utviklet av Kent Bech, Apple Computer, og presentert på konferansen OOPSLA (*Object Oriented Programming, Systems, Languages and Applications*) i 1989. Det ble laget som et undervisningsmiddel. Senere har man funnet at det er nyttig også i andre sammenhenger og metoden er tatt i bruk til analyse og design av objektorienterte systemer.

Kortene

CRC-kort er små kort (det anbefales kort noe mindre enn A5 størrelse) der man kan skrive ned klasse, evt. metaklasser, ansvar og samarbeidende klasser. Kortene kan være helt blanke, så deltakerne kan fylle ut helt fritt selv, eller ha et skjema f.eks. slik:

CRC-KORT

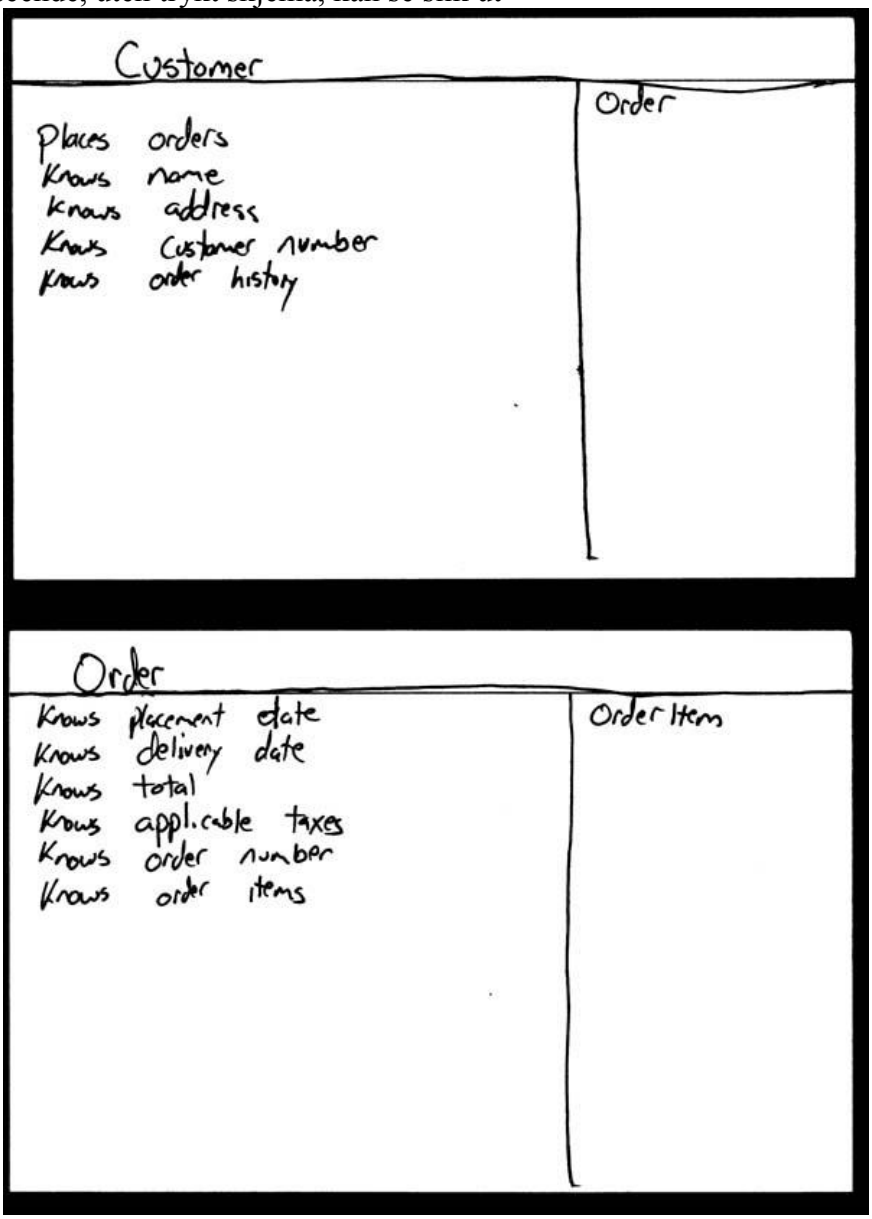
(C) Klasse:	
Metaklasse(r):	
(R) Ansvar (vet og gjør = medlemmer)	**(C) Samarbeider med (assosiert med)**

De forskjellige begrepene kan beskrives slik:

- ✓ Klasse: Navnet på en klasse, f.eks. Kunde, Ordre, Faktura. Kortet representerer da klassen. Noen gjør det slik at når kortet ligger nede på bordet, representerer det klassen. Når det holdes opp i luften, representerer det et objekt av denne klassen. (Dette kan jo bli litt finurlig.)
- ✓ Metaklasse: Er tatt med av hensyn til arv. Alt ansvar og alle samarbeidsklasser i metaklassen gjelder jo også for denne klassen.
- ✓ Ansvar: Alt objektene i denne klassen vet og gjør. Dette tilsvarer altså objektmedlemmene, men skrives helt uformelt og uten spesiell syntaks.
- ✓ Samarbeider med: Klasser (egentlig objekter) som denne klassen samarbeider med for å få noe gjort, eller henter data fra. I praksis vil det være andre objekter som dette objektet kan navigere til (gjennom en assosiasjon).
- ✓ Det er ingen sammenheng mellom ansvar og samarbeid på samme rad.

- ✓ Noen anbefaler at hvis man ønsker det, kan man skrive medlemmene mer formelt på baksiden av kortet.

Et enklere utseende, uten trykt skjema, kan se slik ut[36]



Deltakere

Det bør være to-tre representanter for brukere, en-to systemerere, en møteleder og en referent. I tillegg kan man ha med et antall observatører, men de skal ikke delta i diskusjonen.

Fremgangsmåte

Man må ha et system som skal utvikles og en god idé om hva systemet skal kunne gjøre. Det kan være basert på bruksmønstre, men gjerne også bare idéer blant deltakerne. Man møtes – det anbefales maks seks deltakere – og tar en funksjonalitet av gangen.

Man begynner med en idédugnad (brainstorming) etter de regler som gjelder for det. Et mål er å identifisere klasser som inngår i løsningen. Klassenavnene skrives på hvert sitt kort. Et tips

[36] http://www.agilemodeling.com/artifacts/crcModel.htm

er å se på beskrivelsen av prosjektet og lete frem substantiver (”Ordre”, ”Kunde”, ”Faktura” osv.). Videre definerer man ansvar – se etter egenskaper, data o.l. og verb (”registrere”, ”sende”, ”ta kontakt” o.l.). Ansvar skrives inn på kortene. Noen anbefaler å vente med attributter, siden de er detaljer (de vil jo gjerne bli skjult) og evt. skrive dem bakpå kortet. Synlige (public) handlinger hører imidlertid hjemme på forsiden. Idédugnaden kan med fordel dokumenteres med ABE-tabellen som er omtalt i slutten av kapittel 3.

Etter idédugnaden, fordeles kortene til deltakerne. Alle bør være ansvarlig for minst ett kort, men kan ha ansvar for flere.

Deretter spiller man igjennom et scenario. Et scenario er det som skjer etter en hendelse som systemet skal reagere på, f.eks. ”kunden Per ringer og bestiller to paller med kunstgjødsel”. Man vet omtrent hva systemet skal gjøre, men en av klassene må reagere på hendelsen. Da holder den som ”har” klassen opp kortet (da er det et objekt) og sier hva som gjøres. Dette vil da være noe denne klassen har ansvar for. Objektet kan kalle på andre (samarbeidende) objekter og svarer i sin tid til den som spurte.

Bare ”vanlig vei” uten feil spilles slik. Med "vanlig vei" tenker man på alt som skjer normalt. F.eks. vil det være helt vanlig at kunden Per (som vil bestille to paller med kunstgjødsel i eksemplet ovenfor) ikke er registrert som kunde tidligere. Brukeren skal da dirigeres innom kunderegistreringsdelen av systemet. Det er da ikke en "feil", men en vanlig hendelse som systemet skal håndtere i dialog med brukeren. Tilsvarende vil det være helt normalt at man ikke har gjødsel på lager – det skal gi en restordre. Man kan tenke på "feil" som noe som fører til at systemet ikke kan gjennomføre transaksjonen. Man simulerer altså bare "solskinnshistorier" som ender godt.

Når en ”bruker” (utenfor systemet) henvender seg til systemet, bør en klasse reagere med å spørre brukeren om mer informasjon. ”Brukeren" kan da svare at det bør klassen vite selv (attributt eller kall).

Fordeler/ulemper

En stor fordel er at det er enkelt og interaktivt – man får *delta*. Deltakerne får da førstehånds erfaring med systemet lenge før det er laget og kan se problemer og muligheter tidlig. De vil også få en viss forståelse for hvordan programmet er bygget opp.

En annen fordel er at det er svært enkelt. Det trengs svært lite utstyr og ingen maskiner. En tilleggsfordel med dette, er at alle kan forstå det – man trenger ikke være IT-ekspert e.l. Den viktigste kunnskapen her er faktisk den brukerne besitter.

Referanser

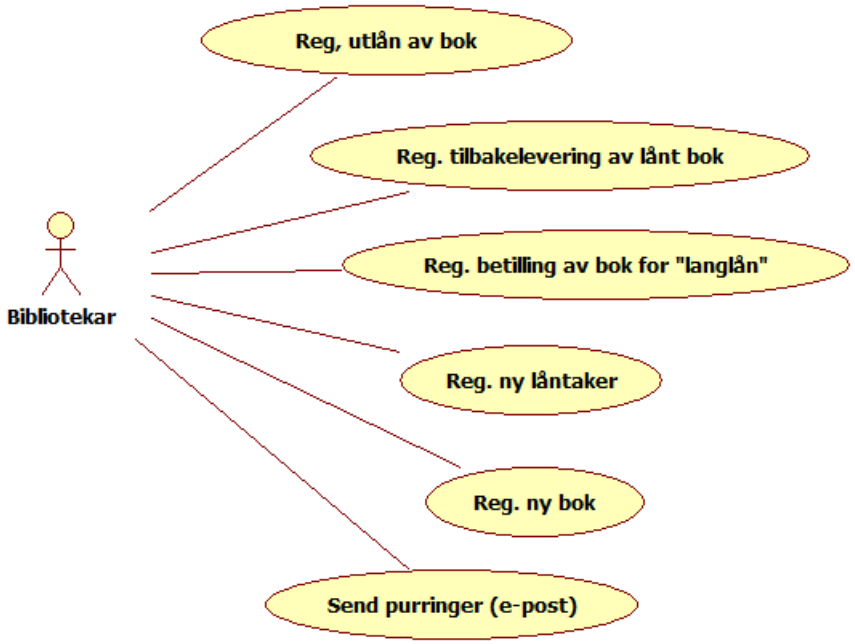
Du finner mer informasjon i disse referansene:

Referanse	Beskrivelse
http://c2.com/doc/oopsla89/paper.html	Den opprinnelige artikkelen til Kent Bech på OOPSLA 1989.
http://alistair.cockburn.us/Using+CRC+cards	En kort ”trinn for trinn” – ”slik gjør du” anvisning
http://www.agilemodeling.com/artifacts/crcModel.htm	Omtale og kort om bruken av CRC

Oppgave/øvelse kapittel 9 (bibliotek)

Det skal lages et utlånssystem for høyskolens bibliotek. Studenter og ansatte skal kunne låne bøker. Normal utlånstid er tre uker, men litteratur som står på en pensumliste kan bare lånes i tre dager. Fagansatte kan låne på såkalt ”langlån”, dvs. at de får beholde boken så lenge der er ansatt ved høyskolen. Til "langlån" kjøpes det alltid inn et eget eksemplar selv om biblioteket allerede har boken.

Det har vært gjennomført en analyse av bruksmønstre, med følgende resultat:



Deres oppgave:

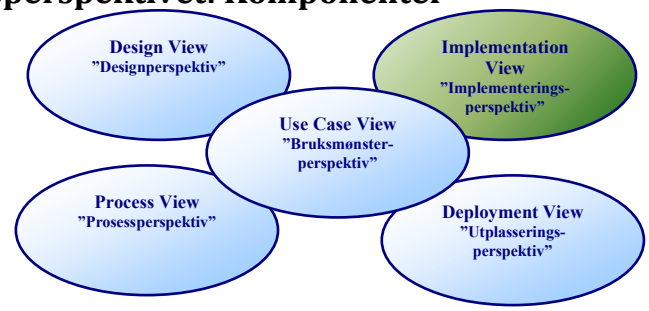
1) Organiser en CRC-seanse (bestem roller).
2) Gjennomfør en idédugnad og finn klasser.
3) Spill følgende fem scenarier:
   a) Student Ole låner boken ”Oles skitur” av Elsa Beskow. Ole er registrert låntaker.
   b) Biblioteket kjører en kontroll, og systemet oppdager at fagansatt Kari har lånt boken ”Kari Trestakk” utover lånetiden. En e-post skal sendes Kari om dette.
   c) Fagansatt Knut ber om å få boken ”OOAD” av D. E. Sign på "langlån". Slike bøker må kjøpes inn særskilt.
   d) ”Snekker Andersen” av Alf Prøysen blir levert i bokkassen utenfor biblioteket utenom bibliotekets åpningstid. (Det vil vise seg at det er student Alf Andersen som leverte og det var innenfor leveringsfristen.)
   e) Student Lat Hans låner pensumboken "OOAD" av D. E. Sign. Dette er en pensumbok. Lat er ikke tidligere registrert som låntaker.

Kapittel 10 – Øvrige views: Implementering, utplassering og prosess

Disse perspektivene har en mer fysisk tilnærming til design og innebærer planlegging av systemet i drift: Hvordan er det da delt opp og hvilke maskiner kjører det på?

Det vil være mest av interesse for store, komplekse systemer som skal gå i et nettverk av klient/tjenere.

Implementeringsperspektivet: Komponenter



Implementeringsperspektivet skal vise hvordan man tenker at systemet skal være delt opp når systemet er implementert (altså laget og i drift).

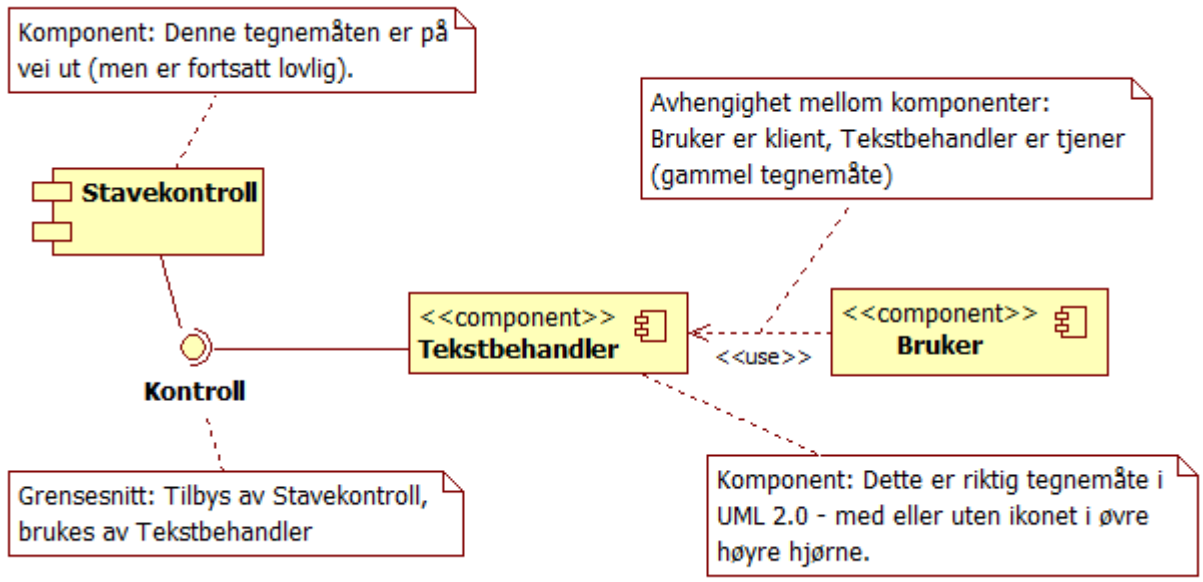
Programmer deles i *kodemoduler* (programmeringsuttrykk) som *kaller hverandre*. Det er ingen diskusjon her om hvilke maskiner de fysisk skal ligge på. En kodemodul som har klart definert grensesnitt til andre moduler, kan byttes ut med en annen modul med samme grensesnitt, på samme måte som en "original del" på en bil, kan byttes mot en "uoriginal del" laget av andre. Slike deler kalles ofte "pluggkompatible" enheter. Fordelen er at slike enheter kan gjenbrukes i andre systemer (bildekk passer på mange bilmerker) og at enheten kan vedlikeholdes separat (sålenge grensesnittet opprettholdes, evt. bare utvides). I UML – og i mange andre sammenhenger – kalles slike enheter *komponenter*.

En komponent i UML var tidligere en fysisk enhet som kunne byttes ut uten at det øvrige systemet måtte endres – en form for "pluggkompatibilitet". I UML 2.0 har man gjort om på dette, og en komponent kan nå også være en logisk enhet som kan anvendes i flere sammenhenger (gjenbruk). UML 2.0 sier det slik:

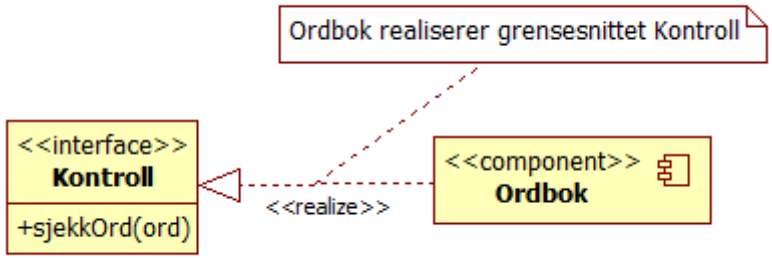
> *The Components package specifies a set of constructs that can be used to define software systems of arbitrary size and complexity. In particular, the package specifies a component as a modular unit with well-defined interfaces that is replaceable within its environment.*
>
> ……
>
> *An important aspect of component-based development is the reuse of previously constructed components. A component can always be considered an autonomous unit within a system or subsystem. It has one or more provided and/or required interfaces (potentially exposed via ports), and its internals are hidden and inaccessible other than as provided by its interfaces.*
>
> *Kilde: Unified Modeling Language: Superstructure, versjon 2.0*

En *komponent* er altså en *samling* klasser – et *delsystem* – som har full informasjonsskjuling med ett/flere ytre *grensesnitt*. Komponenten er en spesiell form for klasse. Det samme er grensesnittene, og begge kan derfor også tegnes som klasser. Komponentklassene vil ikke ha attributter/metoder men derimot tilgjengelige grensesnitt. Grensesnittene vil ha tilgjengelige metoder.

Den vanligste notasjonen fremgår av nedenstående figur:

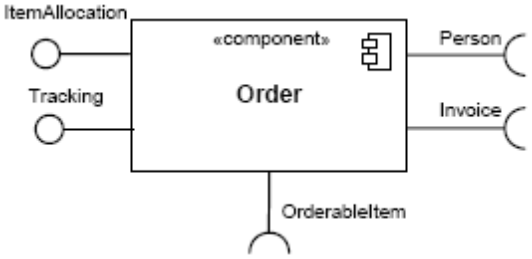


Denne figuren har med de vanligste symbolene.



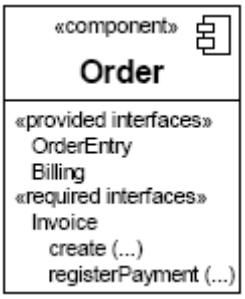
Her er grensesnittet Kontroll fra figuren ovenfor vist som en vanlig klasse, med operasjoner. Grensesnittet skal realiseres (dvs. håndteres) av komponenten Ordbok.

Det finnes flere diagramelementer enn de som er vist ovenfor, f.eks.[37]

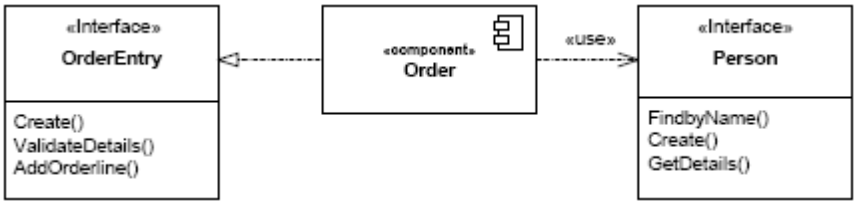


En komponentklasse: Sirklene er "tilbudte grensesnitt" (som denne klassen tilbyr andre), halvsirklene er "nødvendige grensesnitt" (som denne klassen trenger).

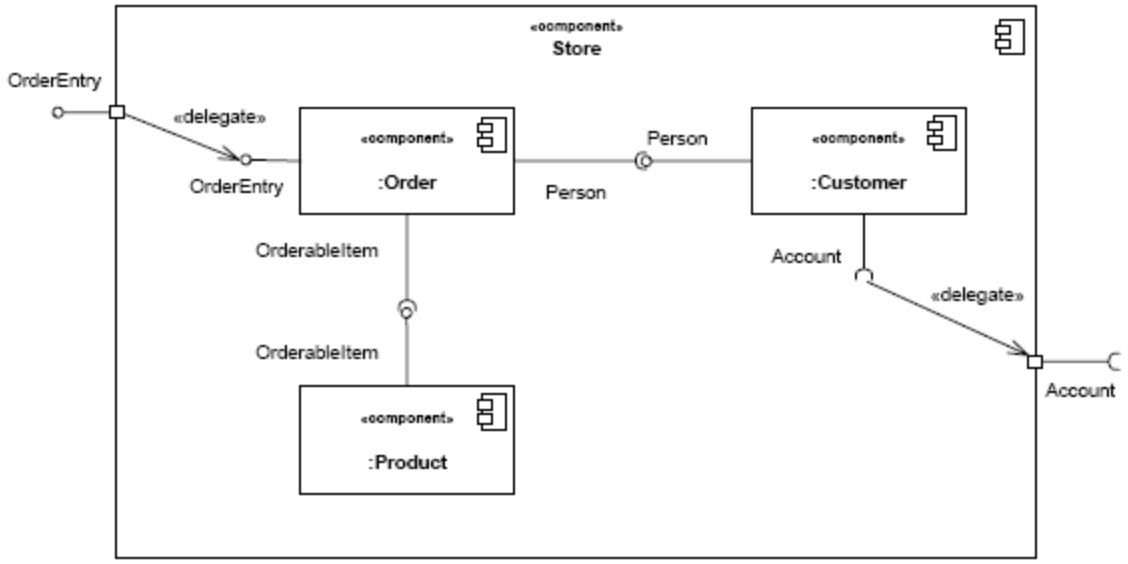
[37] Hentet fra spesifikasjonen av UML 2.0: Unified Modeling Language: Superstructure, version 2.0



Her er klassen vist på vanlig måte, med grensesnitt (ikke med medlemmer)

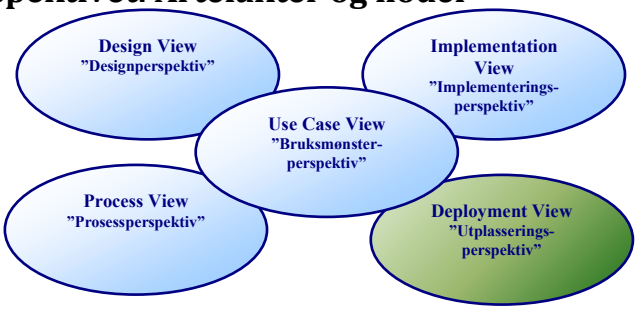


Her er også grensesnittene vist som klasser



Her er flere komponenter lagt sammen til en overordnet komponent med ytre grensesnitt. Man ser at f.eks. det ytre grensesnittet "OrderEntry" får kontakt med det "indre" grensesnittet med samme navn, gjennom en "delegate" (signalene kommer til det eksterne grensesnittet, som delegerer til ett eller flere indre grensesnitt som til sammen utgjør det ytre grensesnittet).

Utplasseringsperspektivet: Artefakter og noder



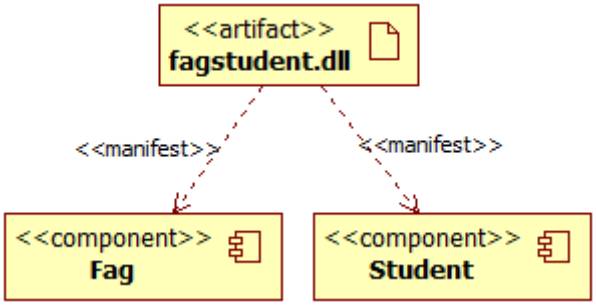
Utplasseringsperspektivet skal vise hvordan man tenker at systemets fysiske manifestasjoner skal installeres på forskjellige maskiner i et nettverk når systemet kjøres.

Artefakt

Artefakter (="ting som er laget av mennesker") er i denne sammenheng fysiske samlinger av bits. De lagres fysisk som filer på disk, f.eks. dokument, datafil, programfil, bildefil, multimediafil osv. (Ordet brukes også om den *dokumentasjonen* som produseres under en systemutvikling med UML, f.eks. alle diagrammene.) UML 2.0 spesifikasjonen sier det slik:

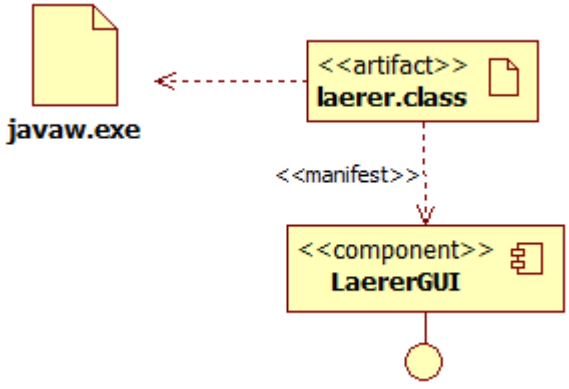
> *An artifact is the specification of a physical piece of information that is used or produced by a software development process, or by deployment and operation of a system.*

Artefaktene er den fysiske representasjonen av komponenter (som igjen er samlinger av klasser – et delsystem). De tegnes som klasser, med stereotypen <<artifact>> og evt. symbol i øvre, høyre hjørne:



Avhengigheten er merket <<manifest>> dvs. manifesterer = "kroppsliggjør" = gjør fysisk. Programbiblioteket *fagstudent.dll* er altså den fysiske representasjonen av de to komponentene *Student* og *Fag* (som igjen sikkert inneholder forskjellige student- og fagklasser).

Det kan være vanlige avhengigheter mellom artefakter (javaw.exe er run-time Java for Microsoft OS og er her tegnet på alternativ måte):



Det kan skilles mellom den generiske klassen og en instans (kopi) ved at sistnevnte understrekes, men vanligvis gjøres det ikke, da sammenhengen vanligvis er kjent.

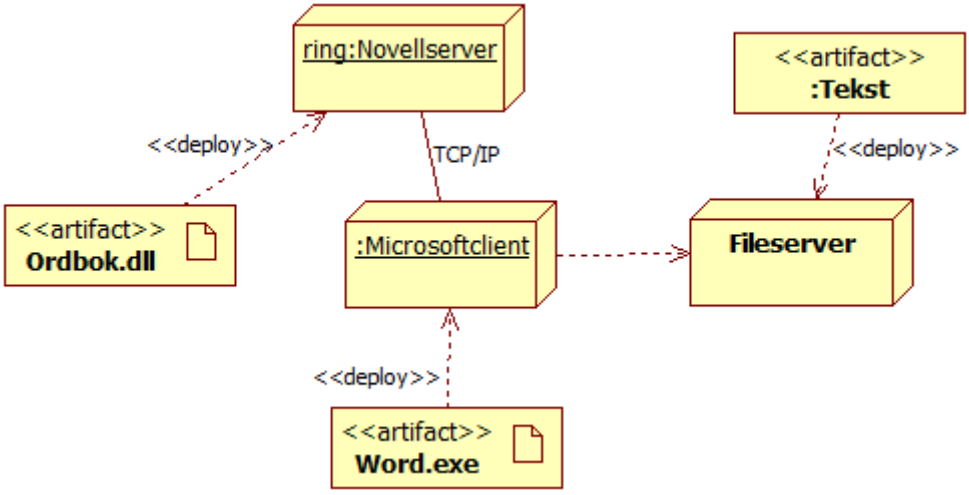
Man kan ytterligere presisere hva slags artefakt det dreier seg om, ved å bruke stereotyper som <<document>>, <<executable>>, <<file>>, <<library>> eller <<source>> som alle er presiseringer av <<artifact>>.

Node

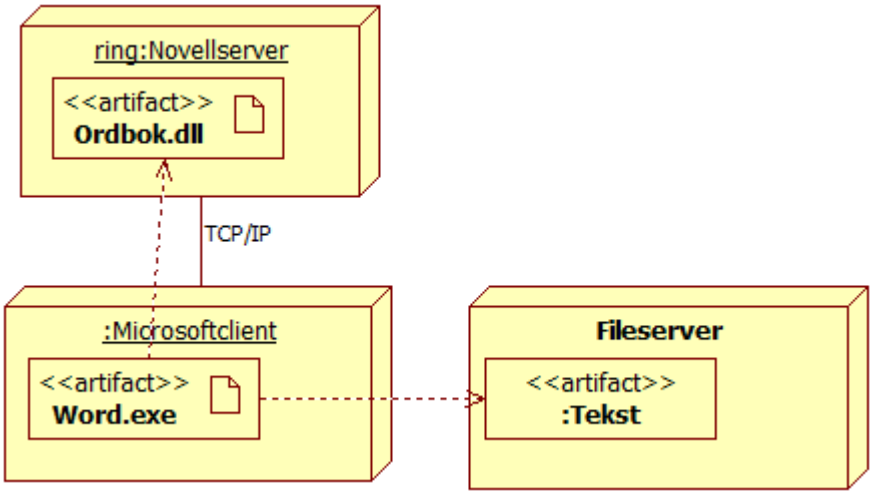
En *node* er en datamaskin eller en enhet som kobler sammen datamaskiner. Noder har alltid minne og kan ha prosessorkapasitet. Komponentene (se kapittel 10) "bor" (eng. *resides*) i en node.

Det skilles mellom generiske noder, her *Novellserver* og nodeinstanser, her *ring*.

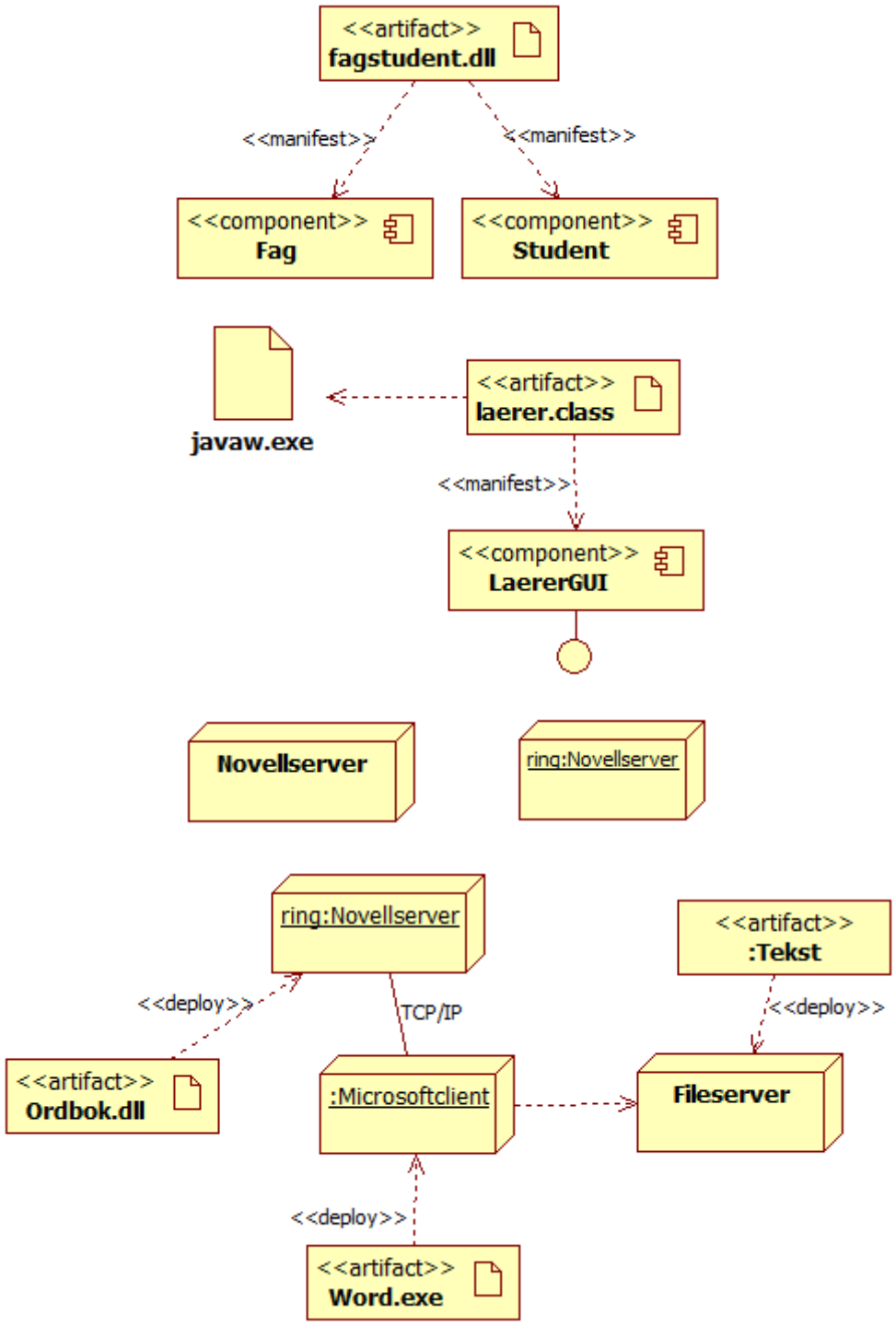
Artefaktene utplasseres (eng. *is deployed*) på noder som kommuniserer med hverandre, dvs. de sender signaler og meldinger (se nedenfor om dette).

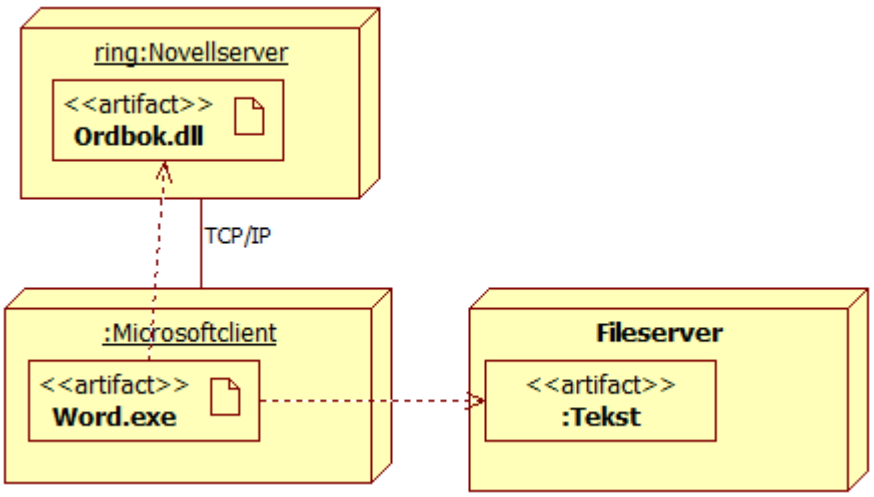


Det er også tillatt med flere andre notasjoner, f.eks. mer grafisk:

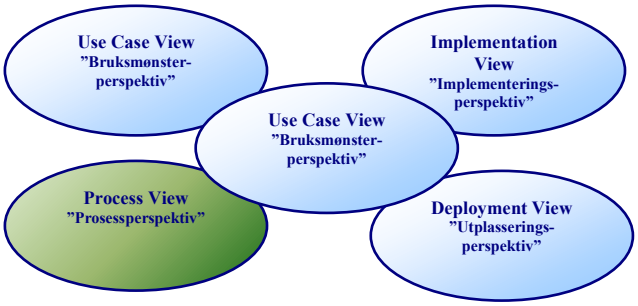


Her får man også tydelig frem hvilke komponenter som er avhengig av hverandre.


<<artifact>>
fagstudent.dll
<<manifest>>
<<manifest>>
<<component>>
Fag
<<component>>
Student
javaw.exe
<<artifact>>
laerer.class
<<manifest>>
<<component>>
LaererGUI
Novellserver
ring:Novellserver
ring:Novellserver
<<artifact>>
:Tekst
<<deploy>>
TCP/IP
<<deploy>>
<<artifact>>
Ordbok.dll
:Microsoftclient
Fileserver
<<deploy>>
<<artifact>>
Word.exe



Prosessperspektivet



I prosessperspektivet ser man spesielt på arbeidsgangen, på flyten av kontroll. Den styres av hendelser i from av objekter som sendes/ankommer og signaler. Innenfor ett system, dreier det seg om parallellitet i synkroniserte tråder, mellom systemer dreier det seg om integrasjon av systemene. For studenter kan et aktuelt eksempel være at studentene registreres i høyskolens studentsystem, FS. Når de melder seg på et kurs i "studweb" – som kommuniserer med FS – får de snart etter tilgang til kursets rom i e-læringssystemet, Fronter. Dette er separate systemer, men de koordineres altså.

Klasse- og objektdiagrammer brukes til dokumentasjon, sammen med interaksjonsdiagrammer (sekvens- og samarbeidsdiagrammer). Videre brukes aktivitetsdiagrammer med en utvidet syntaks for å vise flyten der flere systemer er med.

Det er reist kritikk mot UML 2.0 på dette punkt. Man mener at UML ikke er rikt nok til å dokumentere slike prosesser godt nok.

Jeg ser ikke nærmere på prosessperspektivet i denne boken[38]. Dette kan gi en oversikt:

[38] Interesserte henvises til en utmerket, men riktignok temmelig akademisk artikkel på http://www.itu.dk/courses/SMD/F2008/TH/EFHT05PAIS.pdf.

	activity diagram	class diagram	object diagram	sequence diagram	structure diagram
actions and control flow	X				
data and object flow	X	X	X		
organizational structure	(X)	X	X		
interaction-centric view				X	
system-specific models	X	(X)			X

Table 5.1 Overview of the different UML diagrams

Engels & al.: Process Modeling Using UML", side 30

Oppgave til kapittel 10 (komponenter og utplassering)

Det skal nå lages et system *Kursvalg* for emnepåmelding med en applikasjon som studentene skal bruke. I første omgang skal systemet foreslå at studenten melder seg på de obligatoriske kursene kommende semester. Videre skal studenten få se en liste over mulige valgkurs. Ut fra dette melder studenten seg på et passende antall kurs og dette skal både registreres på studenten og på kursene (som derved har fått en ny deltaker).

Det er tidligere laget en serverapplikasjon for registrering av alle studenter ved høyskolen, med personlige opplysninger som studentnummer, navn, adresse osv. samt hvilket studieprogram/år studenten følger (de er inndelt etter startår). Applikasjonen heter *Studreg* og tilbyr et grensesnitt kalt *Stud_IO* som er tilgjengelig for andre applikasjoner som kobler seg til. *Studreg* kjører på maskinen *Hone2* som er en Windows server og dataene lagres i en Microsoft SQL Server database.

Videre finnes det også en serverapplikasjon der samtlige emner og studieprogrammer er registrert med opplysninger som emnekode, emnenavn, studiepoeng osv. og tilsvarende for studieprogrammene. Når et emne undervises kalles det et kurs og kursene er også registrert der med emnekode, semester osv. Denne serverapplikasjonen kalles *Kursreg* og har grensesnittene *Emne_IO* og *Kurs_IO* som også er tilgjengelig for alle. *Kursreg* kjører på en LAMP[39] server kalt *KongA*.

Den nye applikasjonen *Kursvalg* skal også kjøre på *KongA* med et nettbasert grensesnitt til studentene.

All kommunikasjon mellom applikasjonene skjer med TCP/IP-protokoll over Intranettet.

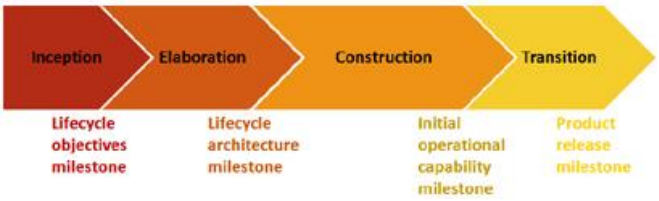
Oppgave A: Bruk StarUML og lag komponentdiagram for dette systemet.

Oppgave B: Bruk StarUML og lag utplasseringsdiagram for systemet.

[39] Linux, Apache, MySQL og PHP

Kapittel 11 – Utviklingsmodell: Kort om RUP (Rational Unified Process)

Note 1: Dette kapitlet er ikke direkte relatert til UML-standarden men stoffet her er viktig for OOP-programmere. RUP er den metoden som "hører sammen med" UML.



Kilde: http://www.velocis.in/company-profile/software-development-process

Note 2: Beste kilde for RUP er nå IBM, som har overtatt Rational. Ellers har Wikipedia en meget god oversikt over RUP på http://en.wikipedia.org/wiki/Rational_Unified_Process.

Bakgrunnsteori

UML

Som nevnt i innledende kapitler, laget de tre ”amigos” i Rational – Ivar Jacobson, Grady Booch og James Rumbaugh – først UML basert på svært mange kilder. Det er derfor navnet ble "Unified Modeling Language".

Som navnet også sier, er UML bare et *språk* til å modellere objektorienterte systemer, men har ingen *metode*. Med metode forstår jeg her en oversikt over arbeidet som skal gjøres og rekkefølgen i en prosess – ”*a series of steps taken to build software”* (Wikipedia). Det kreves en metode bl.a. for å kunne planlegge arbeidet. Rational fortsatte derfor med å utvikle en slik metode og resultatet ble *RUP*. Arbeidet med RUP ble ledet av professor Philippe Krutchen men de tre "amigos" var fortsatt med og hadde nok stor innflytelse.

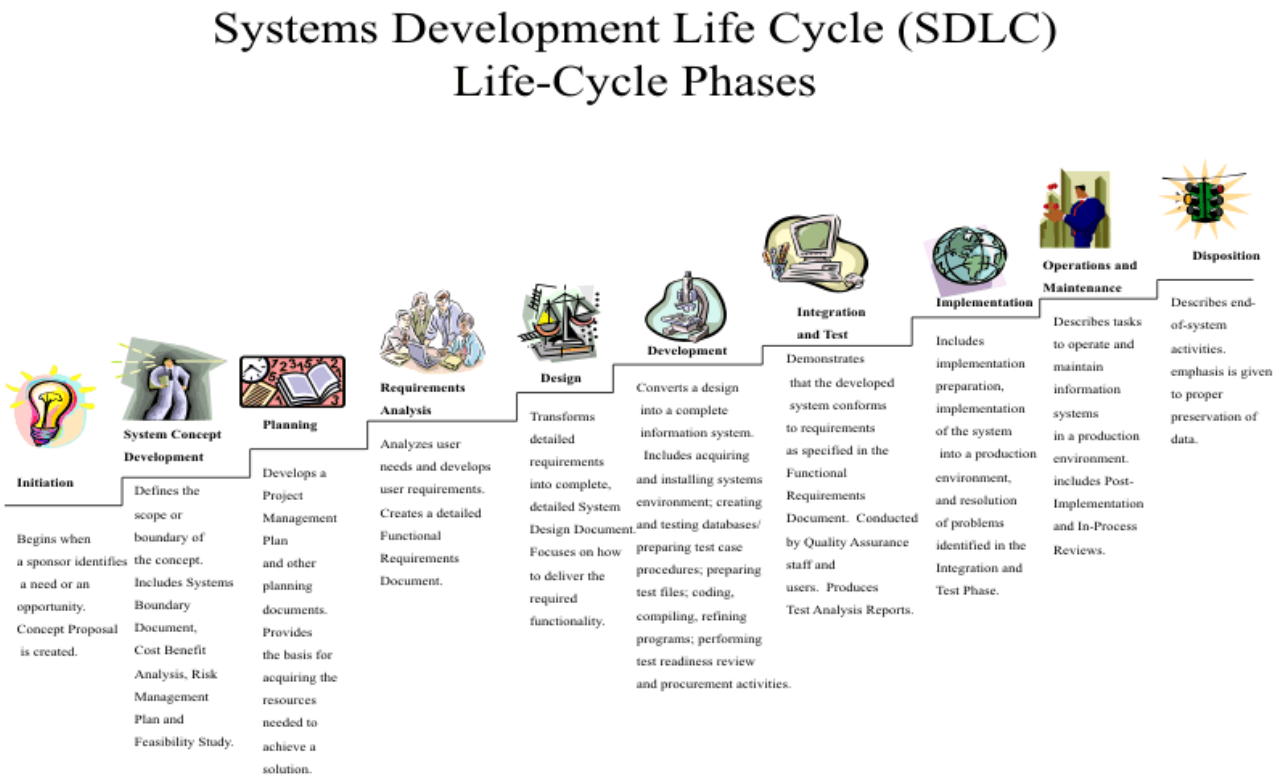
Også RUP bygger på en rekke tidligere arbeider av mange metodeeksperter, og samler dem til én beskrivelse.

Faser

Faser er et gammelt begrep i systemutvikling. De første faseinndelte metodene var sekvensielle, som f.eks. i forskjellige varianter av ”*fossefallsmodeller”* eller *SDLC*. En vanlig modell kan gjengis med akronymet ”FAURIDA”:

1) Forandringsanalyse (hva er problemet og hva bør gjøres)
2) Analyse (dele opp og finne sammenhenger)
3) Utforming (design)
4) Realisering (lage)
5) Implementering (ta i bruk)
6) Drift/vedlikehold (sikker, stabil drift og mindre endringer ved behov)
7) Avvikling (ta vare på data og gjenbrukbare komponenter)

Her er en annen og mer detaljert fra det amerikanske justisdepartement:



Kilde: US Department of Justice (2003): "INFORMATION RESOURCES MANAGEMENT", kap 1

Hver fase avsluttes med å levere noe, som så danner grunnlaget for arbeidet i neste fase. I realiteten er ikke arbeidet tenkt fullt så sekvensielt som det er gjengitt – man kan (1) gå tilbake (iterasjon), (2) jobbe litt med en senere fase (look-ahead) og (3) avbryte arbeidet.

Litt hånlig kalles slik utvikling også BMUF = ”Big Modeling Up Front”. Når arbeidet er ferdig, og leveres, skjer det en *revolusjon* der alt endres på en gang.

Milepeler

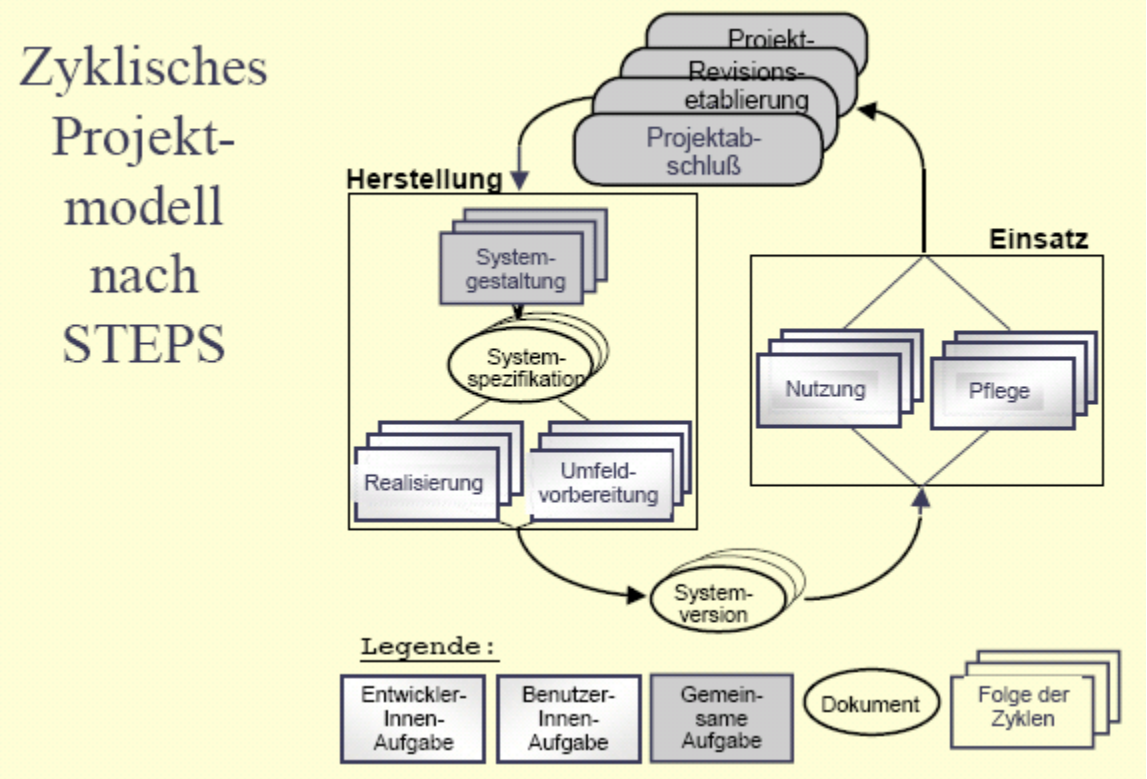
Milepeler er blant annet innført gjennom det norske ”Målrettet prosjektstyring”[40]. En milepel er en *tilstand*, noe som er oppnådd. Begrepet er hentet fra de pelene som sto langs veiene og anga hvor langt man var kommet og hvor langt det var igjen. I utlandet var de helst av sten – derav det engelske navnet.

Kilde: http://herdingcats.typepad.com/my_weblog/2010/03/milestones-resist-milestones.html

[40] Se f.eks. E. S. Andersen & al.: "Målrettet prosjektstyring", NKI-forlaget.

Evolusjonær systemutvikling

En rekke teoretikere har understreket betydningen av å gjenta arbeid, gjerne med stadig større grad av forfining. Én slik teoretiker var professor Christiane Floyd, Berlin, som laget metoden STEPS. Floyd var opptatt av reell brukermedvirkning, som hun mente bare kunne oppnås gjennom faktisk bruk av systemet. Hun anbefalte derfor at man raskt leverer en versjon til brukerne, som anvender det og gir tilbakemeldinger. Dette er en form for evolusjonær systemutvikling.

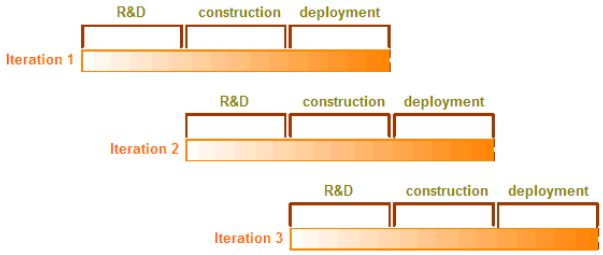


Herstellung = produksjon. Einsatz = virkning. Pflege = vedlikehold. Nutzung = bruk.
Umfeldvorbereitung = forberedelse av organisasjonen som systemet skal brukes i.
Kilde: http://pigseye.kennesaw.edu/~dbraun/csis4650/A&D/UML_tutorial/history_of_uml.htm

I STEPS leveres *hele* systemet, men i uferdig versjon første gang. For hver iterasjon blir systemet bedre inntil brukerne er fornøyd.

Inkrementell systemutvikling

Når et system leveres inkrementelt, leveres bare en del av gangen. Delen må naturligvis velges slik at den har mening, og iflg. praktiker og teoretiker Tom Gilb, bør man hele tiden velge å lage/levere den "saftigste" delen av det som gjenstår (”the juiciest bit”). På denne måten kommer man raskt i gang med bruk, og kan overbevise brukerne om at man faktisk holder på med noe nyttig. Det argumenteres da med at lang analyse, utforming og realisering gjør brukerne trette og fører til en revolusjon når systemet endelig er helt ferdig.



Kilde: http://www.bolour.com/blog/

Prototyping

En *prototyp* er en modell som kan prøvekjøres. Fordelen er at brukere og teknikere kan se systemet "i arbeid", og teste det ut. Det skal ikke brukes for mye arbeid på å produsere prototypen.

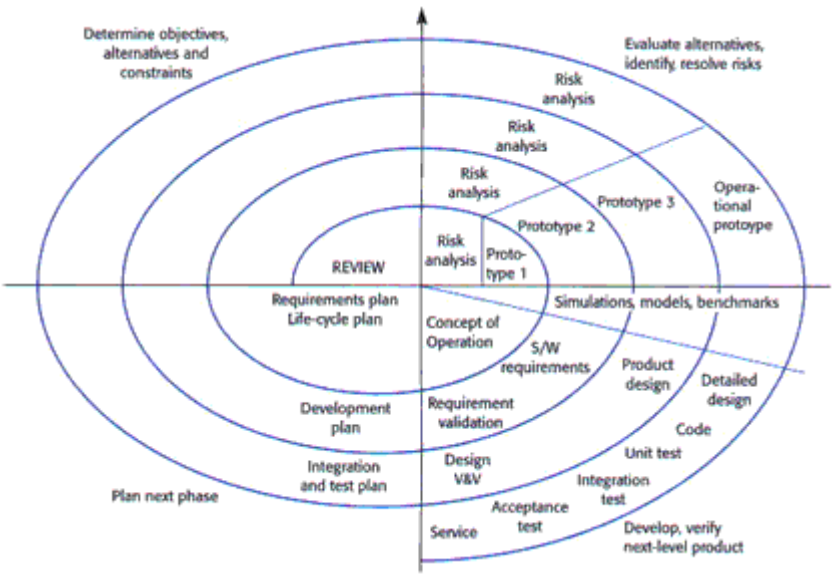
Man har prototyper som utvikler seg til det ferdige systemet, men det vanligste er at man bare vil teste ut noe (grensesnitt, programkode, mange brukere, responstid, kommunikasjon o.l.). De siste skal gjerne kastes etterpå, og kalles derfor "throw away prototypes". Man kan lage prototyper vertikalt (så det virker 100 %, men bare for en del av funksjonaliteten) eller horisontalt (mye/all funksjonalitet er tatt med, men bare overfladisk). En spesiell type horisontal prototyp for brukergrensesnittet, kalles "mock up". Da ser alt ut til å virke, men i virkeligheten lagres intet og skjermbildene genereres ikke – de bare hentes fra fil.

Syklisk systemutvikling

Syklisk (av gr. *cycle* – noe som går i sirkler - som hjulene på en sykkel). En metode er syklisk når den gjentar noe om og om igjen – i sirkler.

Boehms spiralmodell

Barry Boehm var spesielt opptatt av systemutvikling under stor usikkerhet, f.eks. fordi det er uklart hva som skal lages, hva som kreves eller om det overhodet er mulig å få det til. I slike utviklingsprosjekter må nærmest ses på som et eksperiment, eller forskning. Da er det viktig å håndtere usikkerheten på en ryddig og grundig måte.



Kilde: http://koolkampus.com/engineering-notes-1/computer-science/the-spiral-model/

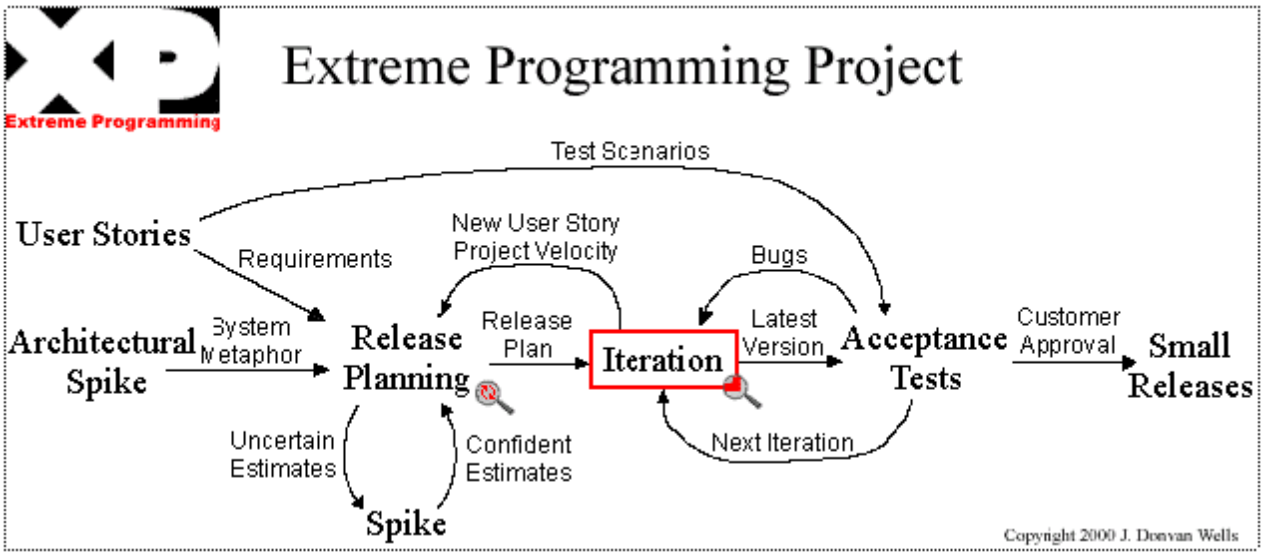
I tegningen er det to dimensjoner: Hvor langt man har "dreiet" angir hvor langt man er kommet med utviklingen, mens avstanden til origo angir arbeidsomfanget. Når man altså stadig dreier om origo og kommer lenger fra origo, oppstår en spiral (REVIEW i figuren).

Spiralmodellen bruker prototyping intensivt for å håndtere risiko. Som det fremgår, er metoden syklisk. Det er lagt inn en hovedbeslutning (om man skal fortsette) for hver syklus.

Extreme Programming (XP)[41]

Dette er en metode som har vært mye omtalt. Den er spesielt opptatt av at brukerne skal tilfredsstilles ved at de får den programvaren de ønsker, når de ønsker den. Metoden skal være fleksibel, også sent under utviklingen. Utviklerne samarbeider tett med brukerne under hele utviklingsprosessen og metoden er derfor gjort så enkel som mulig (hevder man☺).

Fire hovedprinsipper ligger til grunn: Kommunikasjon, enkelhet, tilbakemeldinger og mot. Leveringer skjer så tidlig som mulig, slik at brukerne kan gi tilbakemeldinger. Metoden er iterativ og evolusjonær/inkrementell.



Kilde: http://www.extremeprogramming.org/map/project.html

Oppsummering – bakgrunnsteori for RUP

De tre "amigos" har anvendt tankegods fra alt det ovenstående – og mer til – når de laget sin modell RUP. Metoden er derfor blitt temmelig kompleks, men – som UML – også svært fleksibel.

RUP anvender følgelig faser, milepeler, evolusjonær/inkrementell levering, prototyper, spiralmodellen og meget annet. Deler av dokumentasjonen lages naturligvis med UML.

Anvendelsen av bakgrunnsteorien i RUP

"Stakeholders"

"Stakeholders" er de som har en interesse i prosjektet – på norsk vanligvis kalt *interessentene* ("to hold a stake in" = "å ha en spesiell interesse av"). Når målene for systemutviklingen skal

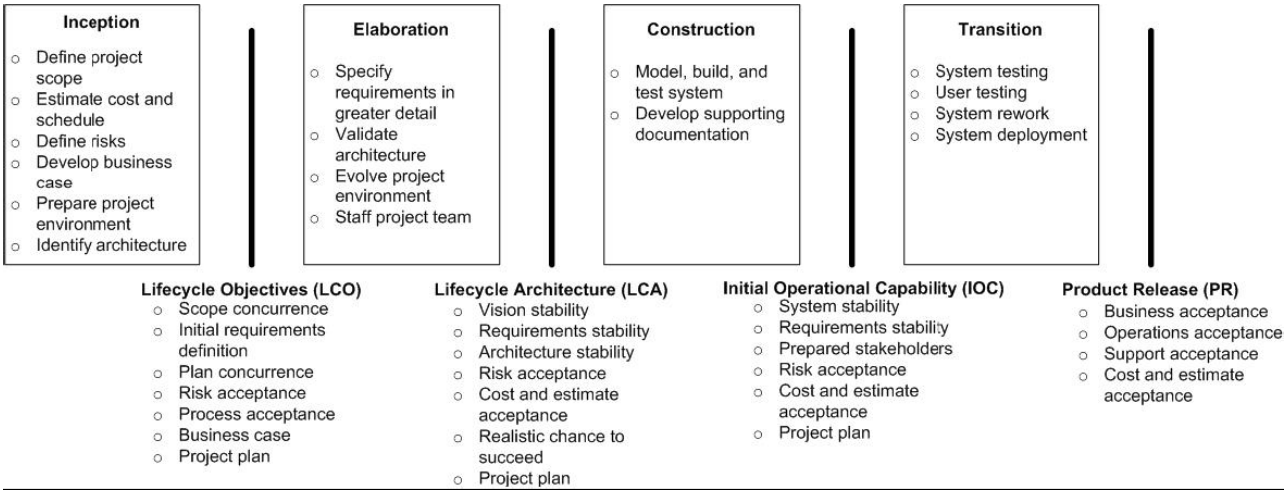
[41] Se http://www.extremeprogramming.org/rules.html. SPIKE er et norsk forskningsprosjekt for forbedring av programvare = Software Process Improvement based on Knowledge and Experience. User Stories er en verbal form for bruksmønstre.

fastlegges, kan det være interessemotsetninger mellom de som skal lage systemet (her kalt ”architects” og ”programmers”), de som skal bruke systemet (”stakeholders”) og kunder, myndigheter osv. (”marketplace”). Det må altså foretas en interesseavveining mellom dem når utviklingsmålene fastlegges, så man får omforenede mål (”objectives”).

Generelt om fasene

De fire fasene er

1) Inception (begynnelse, oppstart)
2) Elaboration (utdypning)
3) Construction (konstruksjon)
4) Transition (overgang)



Kilde: http://www.enterpriseunifiedprocess.com/essays/phases.html

Målet for en fase er alltid er å *levere noe*. Det som leveres kalles ofte *”deliverables”*, og leveringen markerer en *milepel*. I forbindelse med avslutningen av en fase, skal også prosjektet *evalueres* og man må ta en beslutning om *arbeidet skal fortsette*. I så fall må også neste fase *planlegges*.

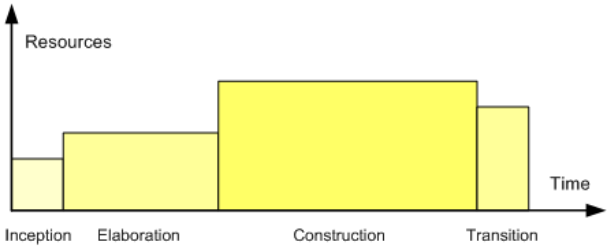
Hvis ikke milepelen nås, må man avgjøre om prosjektet skal avsluttes (uten å bli ferdig), eller om man skal iterere og arbeide med fasen en gang til.

Arbeidsinnsatsen for fasene kan typisk fordele seg omtrent slik:

	Initiation	Elaboration	Construction	Transition
Effort	10%	30%	55%	5%
Elapsed time	20%	35%	40%	5%

Kilde: https://en.wikibooks.org/wiki/RUP_-_IBM_Rational_Unified_Process/Phases

Det vil imidlertid selvsagt variere meget sterkt etter type utviklingsprosjekt, slik at det f.eks. kan ta mye tid i Inception hvis prosjektet skal pløye ny mark teknologisk eller forretningsmessig, mens utvikling innenfor tradisjonelle og kjente områder vil ha hovedvekten på Construction.

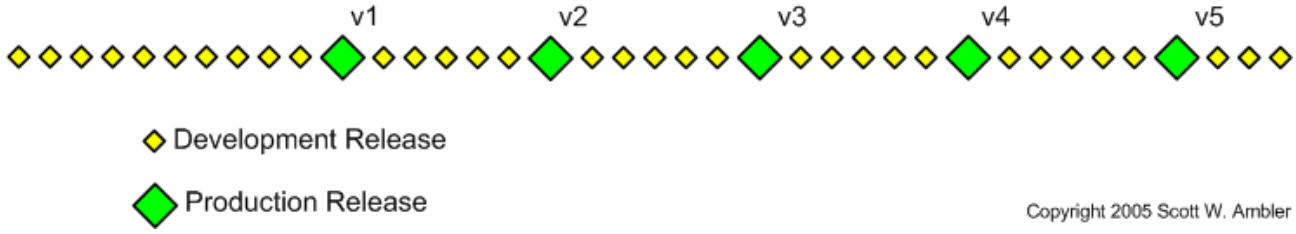


Kilde: https://en.wikipedia.org/wiki/Unified_Process

Fasene kan gjentas så mange ganger som nødvendig, og derved er både evolusjonær og inkrementell systemutvikling mulig.

Inkrementelt/evolusjonært

Med RUP er det lagt opp til levering av fullstendige produkter i stadig bedre versjoner, kalt "Product Release". Avhengig av hvor ferdig produktet er, kan det være en intern "alpha-versjon" eller versjon 1.0. Deretter går man gjennom de fire fasene helt eller delvis igjen, og leverer neste del med nytt versjonsnummer osv.



Kilde: http://www.ambysoft.com/unifiedprocess/agileUP.html

Hvis det passer bedre, kan man også – iflg. RUP – levere deler av funksjonaliteten og så fylle på med mer funksjonalitet i neste syklus (som i figuren vist ovenfor i avsnittet "Inkrementell systemutvikling").

Prototyper

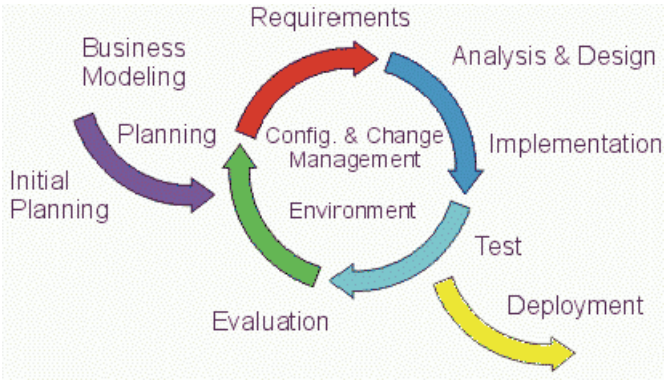
RUP anbefaler prototyper av alle typer, avhengig av formål.

Iterativt

Innenfor hver fase, er det lagt opp til iterasjoner. Etter hver iterasjon kommer man nærmere målet for fasen. Det er ingen regel for hvor mange ganger man skal iterere innenfor en fase, men det er vanlig å tenke at det er færrest iterasjoner under Inception.

Risikohåndtering

Som Boehm, er også RUP svært opptatt av risikohåndtering. Man ser at alle de tre første fasene nevner risiko spesielt. Arven fra Boehm blir særlig tydelig i denne figuren:



Kilde: http://www.qrst.de/wiki/rational_20unified_20pr.html

De enkelte fasene i RUP

Inception

Under denne fasen er man opptatt av å få et omforent syn på hva som skal lages. Merk at målet er *enighet* mellom utviklerne og kunden, ikke nødvendigvis *full kunnskap* om det nye systemet – den kommer litt etter litt. Mange *bruksmønstre* (halvparten? en tredjedel?) identifiseres.

Det er videre behov for overordnet planlegging av prosjektet, herunder kostnadsestimering, identifisering av risiki, prioritering, nedsetting av prosjektgruppe, kvalitetskrav osv. Hvis

kunden er erfaren og har mye IT-kunnskap, kan dette være fort gjort, hvis kunden er uerfaren og/eller har lite IT-kunnskap, kan det ta lang tid og kreve mye arbeid.

Milepelen for denne fasen kalles ”Lifecycle Objective Milestone”.

Elaboration
Her skal systemet *planlegges* i detalj og sjekkes. Her skal minst 80 % av bruksmønstrene på plass. Med klassediagrammer og datamodeller beskrives systemet og prototyper lages for de viktigste bruksmønstrene. Prosjektplanen gjøres helt ferdig.

Milepelen for fasen kalles ”Lifecycle Architecture Milestone”.

Construction
Med konstruksjon forstås primært utforming og realisering (dvs. programmering, databasekonstruksjon osv.) så systemet virker slik det skal. Systemet deles i komponenter.

Den tilhørende milepelen er ”Initial Operational Capability Milestone”.

Transition
Fasen tar sikte på å få brukerne til å overta systemet og ta det i bruk. Dette krever blant annet betatesting, opplæring, innkjøp av nødvendig utstyr og overgang fra evt. tidligere systemer.

Milepelen heter ”Product Release Milestone” og dermed er denne syklusen slutt. Deretter kan det være nødvendig å gjennomføre en ny syklus for å produsere neste del (inkrementelt) eller en ny versjon (evolusjonært).

Disipliner innenfor fasene
Innenfor hver fase, jobbes det innenfor såkalte *disipliner*. RUP definerer følgende hoveddisipliner:
1) Business Modeling
2) Requirements
3) Analysis & Design
4) Implementation
5) Test
6) Deployment

Dette likner jo svært på en fossefallsmodell, men det gjelder altså bare innenfor én fase. Se også figuren under "Risikohåndtering" ovenfor.

Det er også nødvendig med visse støttedisipliner:
7) Configuration and Change Management (konfigurasjonsstyring = hold alle dokumentene à jour i takt)
8) Project Management
9) Environment (verktøy, arbeidsplasser o.l.)

Når man altså f.eks. skal gjennomføre en iterasjon av Elaboration, må man drive med litt Business Modeling, noe Requirements, mye Analysis & Design, mye Implementation, og litt av de øvrige disiplinene.

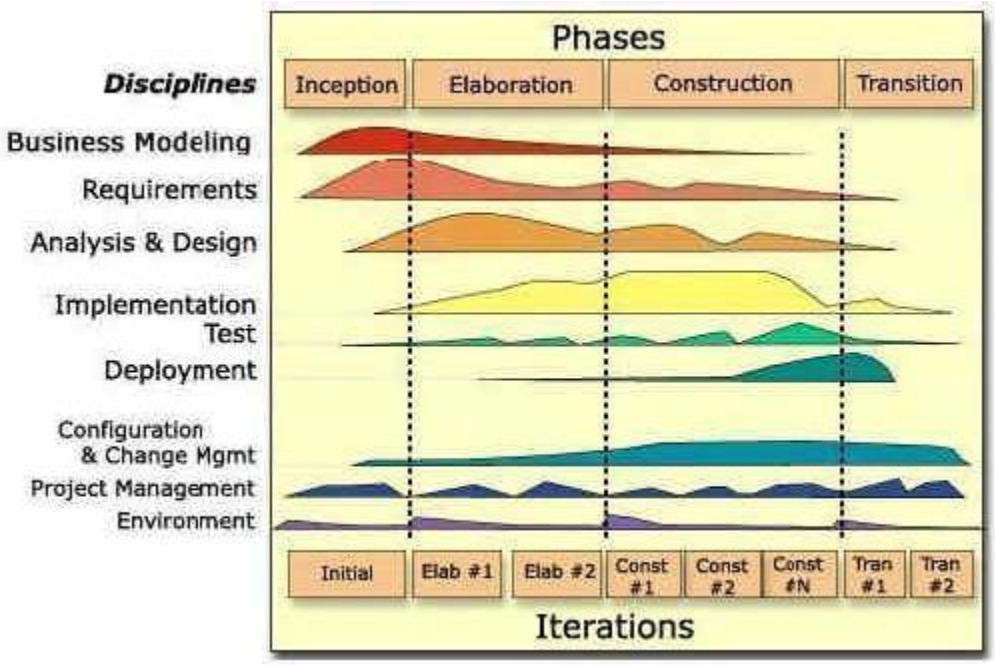
RUP og UML
Det er ikke særlig klart hvordan RUP mener at UML skal benyttes (hvilke faser/disipliner). UML må nok knyttes til disiplinene og ikke til faser. Da kan man anta at det blir behov for

bruksmønstre under Business Modeling og Requirements. Videre er klassediagrammer med tilhørende sekvensdiagrammer, samarbeidsdiagrammer og tilstandsdiagrammer svært aktuelle under Analysis & Design. Komponentdiagrammer og utplasseringsdiagrammer blir mest aktuelt under Implementation og Deployment.

Arbeidsflyt på tvers av fasene

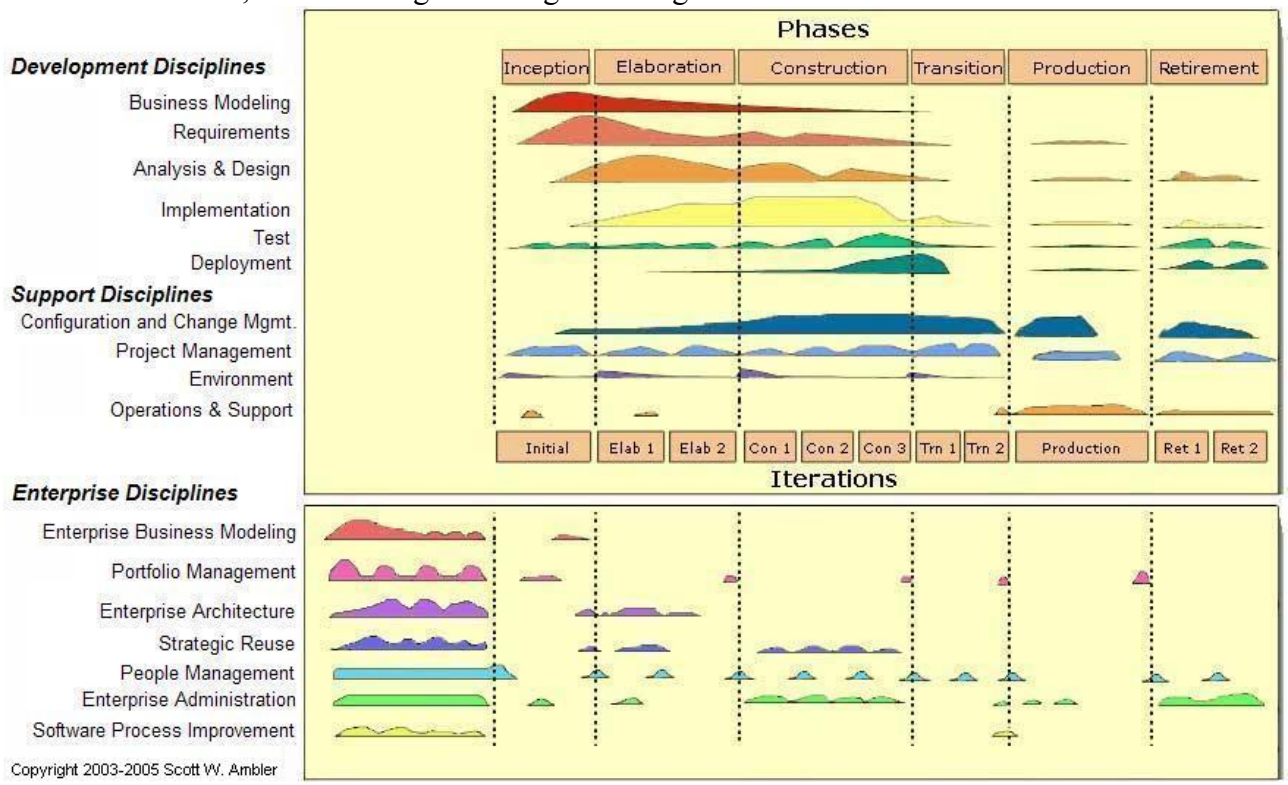
På tvers av alle fasene – horisontalt i modellen – arbeides det i såkalte Work Flows. For hver slik Work Flow er det beskrevet krav til den som skal gjøre det (”role”), hva som skal produseres (”work product”) og hvordan arbeidet kan oppdeles i små, men meningsfylte deler (”tasks”).

Intensiteten i arbeidet varierer gjennom fasene. For Business Modeling, ser man f.eks. at det arbeides relativt mye med det under Inception og Elaboration – senere blir det mindre og mindre. Deployment blir det derimot mer og mer av, ettersom fasene gjennomgås.



Kilde f.eks. P. Kruchten: "What is the Rational Unified Process", Rational Software Canada

I en utvidet modell, ser man at også drift og avvikling er tatt med:



Kilde: http://www.agilemodeling.com/essays/agileModelingRUP.htm

I tillegg er det her tatt med noen virksomhetsdisipliner (”Enterprise Disciplines”). Dette er aktiviteter som hovedsakelig gjøres uavhengig av prosjektene. Det dreier seg om *aktiv forvaltning* av alle virksomhetens systemer. For hvert prosjekt gjøres det litt av disse disiplinene også. F.eks. ser man litt Enterprise Architecture i de tidlige fasene av hvert prosjekt – primært under Elaboration.

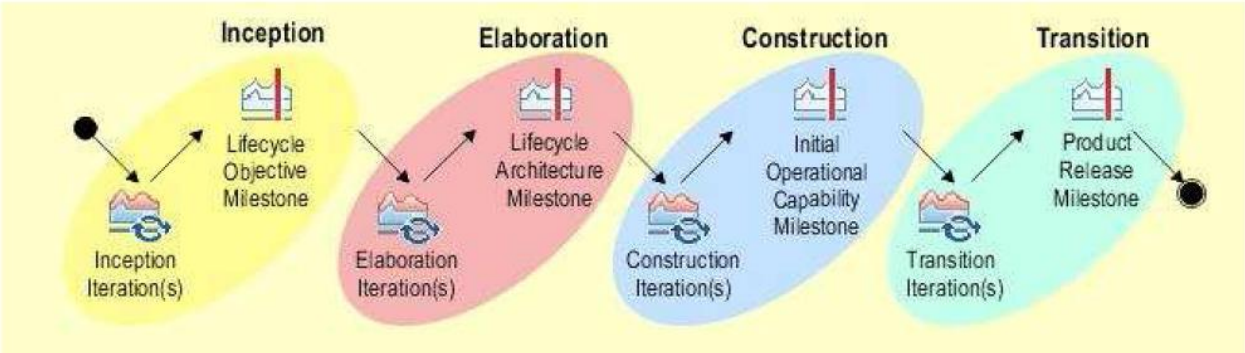
Det er laget fri programvare for RUP, kalt OpenUP ("Open Unified Process") – en del av Eclipse prosjektet. Les evt. om det på http://www.eclipse.org/epf/general/OpenUP.pdf. Programmet skal hjelpe til med planlegging og oppfølging av et RUP-prosjekt. Jeg har ikke prøvd det.

OpenUP beskriver arbeidsprosessen og målene for hver fase slikslik:

Iteration template patterns	Phase objectives
Inception Phase Iteration Initiate Project Plan and Manage Iteration Identify and Refine Requirements Agree on Technical Approach	▪ Understand what to build ▪ Identify key system functionality ▪ Determine at least one possible solution ▪ Understand the cost, schedule and risks associated with the project
Elaboration Phase Iteration Plan and Manage Iteration Identify and Refine Requirements Define the Architecture Develop Solution Increment Test Solution Ongoing Tasks	▪ Get a more detailed understanding of the requirements ▪ Design, implement, validate, and baseline an Architecture ▪ Mitigate essential risks, and produce accurate schedule and cost estimates
Construction Phase Iteration Plan and Manage Iteration Identify and Refine Requirements Develop Solution Increment Test Solution Ongoing Tasks	▪ Iteratively develop a complete product that is ready to transition to its user community ▪ Minimize development costs and achieve some degree of parallelism
Transition Phase Iteration Plan and Manage Iteration Develop Solution Increment Test Solution Ongoing Tasks	▪ Beta test to validate that user expectations are met ▪ Achieve stakeholder concurrence that deployment is complete

Kilde: http://epf.eclipse.org

Det er ganske vanlig i prosjekter å se etter "deliverables". Milepeler knyttes til at et dokument er levert. Det leverte dokumentet utgjør da en "delleveranse" som viser fremdrift, at noe er ferdig og konsulenter kan sende regning for arbeidet som er gjort så langt. OpenUP viser leveringene i denne figuren:



Kilde: http://epf.eclipse.org

I hver fase er det først iterasjoner inntil fasen er ferdig, dokumentasjon leveres og en milepel nås.

For den som vil se mer om RUP i detalj, kan jeg anbefale Agile UP ("Agile Unified Process) som er en forenkling av den mer omfattende IBMs RUP. Du finner den i detalj på

http://www.ambysoft.com/downloads/agileUP.zip. Den er også et godt tips for den som i senere jobbsammenheng skal lage en plan for et litt større prosjekt.

Oppgave til kapittel 11 (RUP)

Oppgave A: Studer forskjellen mellom den tradisjonelle fossefallsmodellen og inkrementell utvikling (med RUP) i denne videoen:
https://www.youtube.com/watch?feature=player_embedded&v=KNu2YcRK02w

Oppgave B: Les IBMs notat om risikoreduksjon med RUP og skriv en oppsummering på ca. én A4-side. Notatet finnes her:
http://www.ibm.com/developerworks/rational/library/1826.html

Oppgave C: Oversett den forenklede figuren nedenfor til norsk. Forsøk deretter å lage den om igjen etter hukommelsen. Forsøk flere ganger til du husker den.

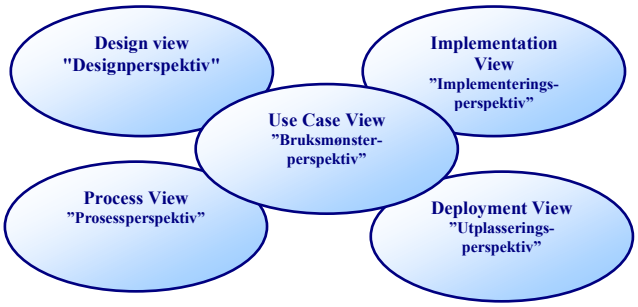
Disciplines	**Phases**			
	Inception	*Elaboration*	*Construction*	*Transition*
Business Modeling	++			
Requirements	++	++		
Analysis and Design		++	+	
Implementation		+	++	
Test		+	+	+
Deployment			+	+
	Initial	*Elab#1, Elab#2,…*	*Const#1, Const#2,…*	*Tran#1, Tran#2,…*
	Iterations			

Oppgave D: Uten å se på figuren, svar på følgende spørsmål:
(a) Hvilke disipliner er de viktigste i fasen *Elaboration*?
(b) I hvilke faser foregår det mest av disiplinen *Requirements*?

Oppgave E: Den nest nederste raden viser *Iterations*. Hvordan kan den også forstås som en modell av *inkrementell levering*?

Kapittel 12 – Oppsummering

Jeg begynte denne boken med en oversikt over system- og systemeringsbegreper og litt bakgrunnsstoff om RUP.



Deretter har jeg gått ganske nøye inn på de forskjellige teknikkene i UML, med særlig vekt på perspektivene *Bruksmønsterperspektivet* og *Designperspektivet*. De andre perspektivene er kortere omtalt.

Jeg har omtalt teknikken CRC som kan bidra i tidlige faser av systemarbeidet og vist hvordan man må "mappe" et objektorientert program til en relasjonsdatabase når man ønsker persistens av objektene.

Til slutt omtalte jeg utviklingsmodellen Rational Unified Process (RUP) som har nær tilknytning til UML bl.a. fordi den opprinnelig ble laget firmaet Rational.

Oppgave til kapittel 12 (oppsummering)

Det er viktig for forståelsen at du danner deg et helhetsbilde av hva denne boken handler om. Et utgangspunkt kan være *innholdsfortegnelsen*. Studer den og vær sikker på at du forstår alle ord som er brukt der, og sammenhengen mellom kapitlene.

Oppgave A: Lag en ordliste med forklaringer av alle faguttrykk i boken.

Oppgave B: Lag en figur som viser sammenhengen mellom kapitlene i boken.

VEDLEGG

Vedlegg 1: Case

Praktisk

Prosjektet kan gjøres individuelt eller i gruppe etter eget valg. Velg en arbeidsmåte som gir maksimal læring.

Bruk StarUML til å dokumentere hele systemet.

"Prosjekter"

En bedrift i Hønefoss har store problemer med å holde orden på timeforbruket til deltakere i systemutviklingsprosjekter, mens kostnadene forøvrig har man god styring på. Derfor vil man nå lage et OO system spesielt for registrering og kontroll av timeforbruket i slike prosjekter. Bedriften bruker en relasjonsdatabase.

I prosjektene deltar både ansatte og eksterne konsulenter. De får en fast rolle i prosjektet, som prosjektleder, dokumentalist, IT-spesialist, brukerrepresentant osv. Rollen kan skifte fra prosjekt til prosjekt og gjennom prosjekttiden, men den ansatte har kun én rolle av gangen pr prosjekt og eventuelle tidligere roller tas ikke vare på. De ansatte tilhører bare én avdeling, men kan delta i flere prosjekter samtidig (og prosjektene behøver ikke gjelde deres avdeling).

Prosjektene kan omfatte én eller flere avdelinger. Man skiller mellom *nyutviklingsprosjekter* og *vedlikeholdsprosjekter*. Alle bedriftens programmer (applikasjonsprogrammer) er registrert, og et vedlikeholdsprosjekt vil være knyttet til ett eller flere eksisterende slike programmer, men hvert program vil aldri bli vedlikeholdt samtidig i flere vedlikeholdsprosjekter.

Alle prosjekter inndeles i de vanlige fasene i SDLC[42] ("fossefallsmodellen"), men kan når som helst stoppes, eller gå tilbake til forrige fase. Hvis det stoppes, er beslutningen endelig. Prosjektet holdes fortsatt lagret i databasen, men noteres som stoppet og kan da aldri settes i gang igjen (hvis arbeidet skal tas opp igjen, må det lages et helt nytt prosjekt).

Det er prosjektlederen som registreres at prosjektet går inn i (eller itererer tilbake til) en ny fase. Det er også prosjektlederen som skriver ut rapporter. Hvis et nyutviklingsprosjekt utvikles ferdig og settes i drift, registreres det som et nytt program. Det kan da *ikke* returnere til forrige fase (I), men må evt. gjøres til gjenstand for et nytt vedlikeholdsprosjekt.

Hvert prosjekt har et budsjett, inndelt i uker, med totalt antall timer som forventes brukt denne uken for alle prosjektdeltakerne under ett. Det er prosjektlederen som registrerer forventet timeforbruk på prosjektet. Faktisk timeforbruk for hver deltaker registreres av prosjektlederen ukentlig. Det skal kunne lages en rapport ("timerapporten") for hvert igangværende prosjekt med samlet, faktisk timeforbruk hittil, timebudsjett hittil og totalt timeforbruk hittil for hver deltaker.

IT-sjefen kan gjøre det samme som prosjektlederne, og i tillegg vedlikeholde alle registre.

[42] Foranalyse, Analyse, Utforming, Realisering, Implementering = "FAURI". D = Satt i drift (ferdig), O = Oppgitt.

Oppgave A: Lag bruksmønsterdiagram for systemet.

Oppgave B: Lag klassediagram. Bedriften bruker standarder for metodenavn og innkapsler objektene maksimalt. Vi regner med at alle klasser har objektattributt som identifiserer objektet (f.eks. *prosjekt_ID*, *program_ID* osv.). Ta også med alle attributter som realiserer assosiasjonene, men ikke standardmetodene.

Oppgave C: Prosjektene gjennomgår faser. Tegn tilstandsdiagram for faseovergangene.

Oppgave D: Det må antakelig finnes en operasjon (metode) for å generere den nevnte "timerapporten". Tegn aktivitetsdiagram for operasjonen.

En mulig funksjonalitet (for prosjektleder) er å registrere en person (allerede registrert som ansatt) som ny deltaker i et prosjekt. Ukebudsjettene berøres ikke. Operasjonen kan ha følgende signatur: *regDeltakelse(person_ID, prosjekt_ID, rolle)*.

Spørsmål E: Tegn sekvensdiagram og samarbeidsdiagram for dette.

Vedlegg 2: Prøveeksamen

Nedenstående prøveeksamen er tilpasset kurset INF251 Objektorientert analyse og design ved Høgskolen i Buskerud og er årlig gitt der.

INF251 Objektorientert analyse og design
BOKMÅLSTEKST

Eksamenstid: 4 timer
Hjelpemidler: "Oversikt over diagramnotasjonen i UML 1.5/2.0" av Knut W. Hansson (vedlagt).
Oppgaveteksten består av 133 sider pluss ett vedlegg. Alle oppgaver skal besvares.
Begynn gjerne besvarelsen av hver oppgave på nytt ark.
Vekting er angitt for hvert spørsmål.

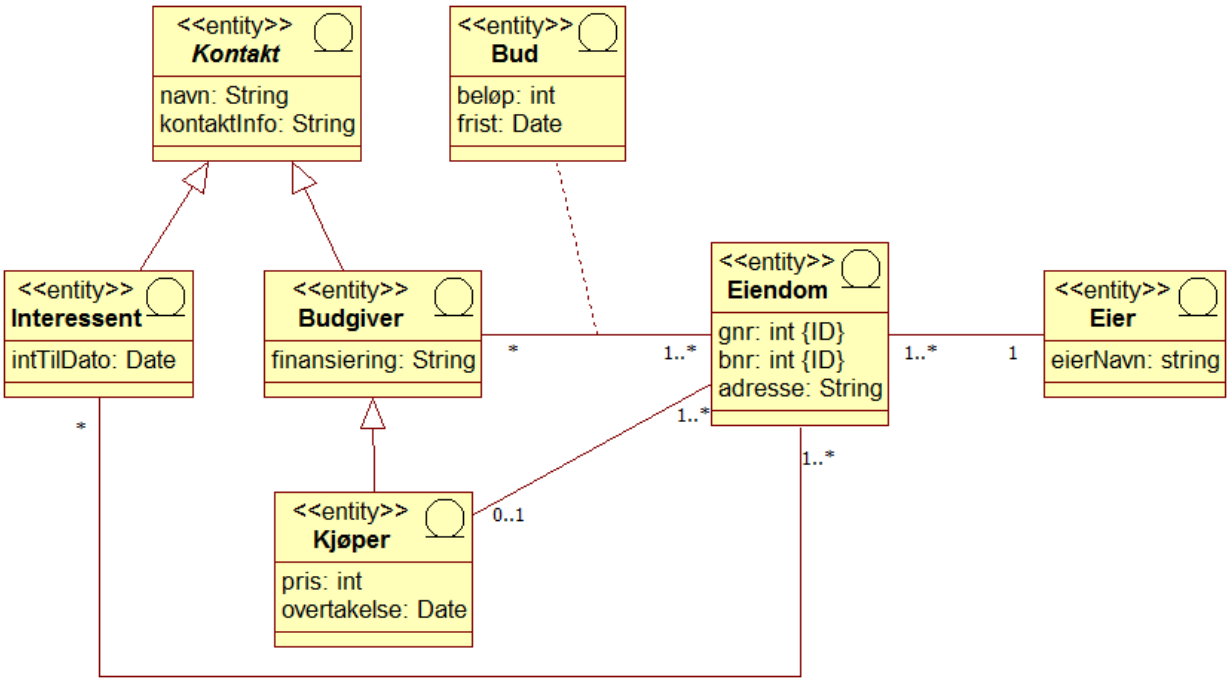
Oppgave 1 – "Relasjonell lagring" – vekt ca 20%

Når man benytter objektorientert programmering, men bruker en relasjonell database, er det nødvendig å tilpasse objektklassene noe, slik at de kan la seg lagre i en relasjonsdatabase. Tilpasningen kan skje i den objektorienterte modellen (så den blir mulig å lagre direkte) eller gjøres i det øyeblikk objektene lagres til databasen.

Spørsmål 1A: Hvilke egenskaper ved objektorienterte programmer er det som gjør at objektene ikke uten videre og direkte kan lagres som rader i tabeller i en relasjonsdatabase?

Spørsmål 1B: Hvilke tilpasninger må gjøres klassene i det objektorienterte programmet?

Gitt nedenstående klassediagram (legg merke til at klassen *Kontakt* er abstrakt og *gnr + bnr* ansees som unik identifikasjon for en eiendom[43]):



[43] Dette er en forenkling – i virkeligheten må også kommunenummer være med, samt evt. festenummer og/eller seksjonsnummer.

Spørsmål 1C: Alle dataene skal lagres i en normalisert relasjonsdatabase. Lag mapping på én av de tre standardmåtene: Filtrert, horisontalt eller vertikalt. [**Vink:** Siden dette er en prøveeksamen, bør du prøve de to andre også, etterpå!]

Oppgave 2 – "Joker'n" (UML) – vekt ca 50%

Baren "Joker'n" har i dag et helt manuelt datasystem. Dagens system beskrives slik:

> Når baren i "Joker'n" er åpen, selges det en rekke varer. Det dreier seg om vin, mineralvann (lettøl, pils og alkoholfrie drikker), sigaretter og snacks. Det er kroassistenter som står i baren og selger. Noen studenter kjøper drinker – dvs. navngitte blandinger – og en drink regnes da som alkoholholdig hvis den inneholder noe alkoholholdig i det hele tatt, uansett styrken.
>
> Kroassistentene registrerer på lister alt som selges i baren, men kjøperen registreres ikke. Etter krav fra bevillingsmyndighetene lages det to lister: Én for alkoholholdige drikkevarer og én for alle andre varer.
>
> "Joker'n" ledes av Jokersjefen. Jokersjefen lager fire ganger i året den såkalte kvartalslisten". Det er en oversikt over kvartalets alkoholsalg pr kveld, fordelt etter type: Øl, vin og alkoholholdige blandinger. Listen inkluderer navnet på Jokersjefen, og sendes til bevillingsmyndighetene.
>
> "Joker'n" har også en innkjøpsansvarlig. Vedkommende sjekker bokført vare-beholdning ved å summere opp salgslistene og trekke denne summen fra forrige, bokførte beholdning. Beholdningene skrives på en ferdigtrykt liste med alle varenavn. Den innkjøpsansvarlige går igjennom listen og merke hver vare med "Tilstrekkelig" eller "Manko" (for lite), etter eget skjønn. Så bestilles de varene som det er for lite av, etter denne listen, og den innkjøpsansvarlige merker de bestilte varene "Bestilt". Når varene ankommer ajourføres beholdningene, og varen merkes "Tilstrekkelig" på varelisten.
>
> Den innkjøpsansvarlige har en oversikt over hvilke leverandører som kan levere varene og det er gjerne flere, alternative leverandører for hver vare. Alkoholholdige varer kjøpes imidlertid alltid fra samme leverandør (men leverandøren varierer fra vare til vare).
>
> Det er Jokersjefen som bestemmer hva baren skal selge (også hvilke drinker), til hvilken pris, og hvem som kan være leverandør(er) for hver vare. Når Jokersjefen bestemmer at en vare skal utgå, merkes den "Utgått" og det kjøpes aldri inn noe mer av denne varen.

Dette systemet skal nå støttes av et objektorientert program og database.

Spørsmål 2A: Tegn de nødvendige bruksmønsterdiagrammene (Use Case Diagrams) for dette systemet.

Spørsmål 2B: Tegn klassediagram. Finn selv på noen få, rimelige attributter i hver klasse. Du kan anta at organisasjonen bruker en standard for navn på tilgangsmetoder og la være å ta dem med i diagrammet. Velg selv passende datastrukturer. Ta med alle attributter som er nødvendig for å realisere assosiasjonene.

Spørsmål 2C: Tegn tilstandsdiagram for varene. Forklar kort sammenhengen mellom tilstandsdiagrammer og klassediagrammer.

Spørsmål 2D: Vis ved sekvensdiagram hva systemet skal gjøre når den innkjøpsansvarlige ønsker å vite beholdningen av alle varer.

Oppgave 3 – "Relasjoner" – vekt ca 15%

I UML er det mange former for *relasjoner*. De har forskjellige navn, forskjellig notasjon og brukes til å angi forskjellige sammenhenger mellom objekter, mellom klasser og mellom andre elementer i modellen.

Spørsmål 3A: Gi en komplett oversikt over relasjonene i UML, med subtyper. Vis notasjonen (hvordan relasjonen tegnes), forklar kort syntaks (hvilke regler som gjelder for bruken) og semantikk (hva den betyr).

Oppgave 4 – "CRC" – vekt ca 15%

En metode for å få med brukerne i diskusjon av objektorienterte systemer, heter "CRC kort" ("CRC cards").

Spørsmål 4A: Hva er CRC kort og hvordan brukes metoden? Hva kan oppnås ved å bruke den?

Slutt på oppgavesettet

Vedlegg 3: Oversikt over diagramnotasjonen i UML

Knut W. Hansson, Høgskolen i Buskerud

Dette er bare et lite utdrag av syntaksen. En *meget* bra og komplett oversikt over notasjonen – med forklaringer – finnes på http://www.uml-diagrams.org/ Oversikten kan benyttes til eksamen i faget INF251 Objektorientert analyse og design.

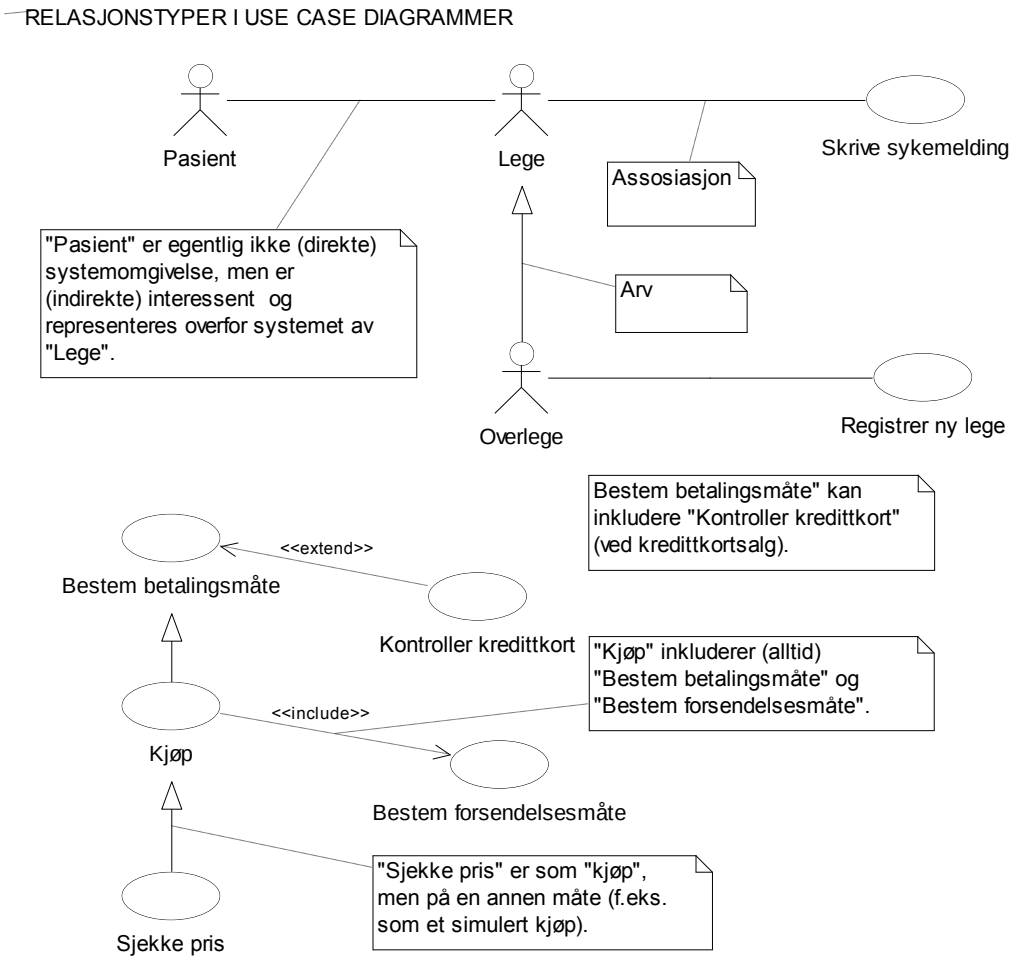
Stereotyper, kommentarer og restriksjoner

Stereotyper (*stereotypes*) er reserverte ord – enten definert i UML eller i brukerprofilen (en standard for prosjektet) – som endrer betydningen av et symbol noe. De skrives i spesielle «anførselstegn» (guillemets) som kan erstattes av <<to mindre - to større>> hvis anførselstegnet ikke er tilgjengelig. Man kan ha flere stereotyper sammen: «metaclass,entity». Stereotyper kan tilknyttes alle elementer i modellen (unntatt andre stereotyper).

Kommentarer (*annotations*) eller **noter** (*notes*) skrives i eget symbol som likner et ark med et hjørne brettet inn – se flere eksempler i grafene her. Kommentaren knyttes til det som kommenteres med prikket linje. En kommentar er ment å klargjøre noe for menneskelige lesere og skrives i vanlig språk. Det som står der endrer ikke noe i modellen, så det er lovlig å skrive hva som helst. Det er vist mange slike i etterfølgende diagrammer.

Restriksjoner (*constraints*) – også kalt **egenskapsstrenger** (*property strings*) – skrives mellom {spissklammer}. De legger beskrankninger på eller innskrenker symbolet de er tilknyttet, f.eks. domenet eller relasjonen. Noen få restriksjoner er predefinerte, f.eks. XOR, men ellers bruker man vanlig språk.

Use Case Diagrams (Bruksmønsterdiagrammer)



Klasser og objekter

{abstract}{persistant/transient}<<entity>>/<<controle>>/<<boundary>> Klassenavn	
+,#,-,~,/	{static}attributtnavn : datatype = initialverdi {lovligverdi1, lovligverdi2,....}
+,#,-,~,/	{abstract}{static}operasjonsnavn (in/out/inout arg1 : arg1datatype = arg1defaultverdi {arg1lovligverdi1, arg1lovligverdi2, ...}, in/out/inout arg2 : arg2datatype = arg2defaultverdi {arg2lovligverdi1, arg2lovligverdi2, ...}) : operasjonsdatatype {operasjonslovligverdi1, operasjonslovligverdi2,....}

- ✓ Abstrakte klasser og medlemmer har navnet i *kursiv*. Bruke gjerne restriksjonen {abstract} isteden.
- ✓ Klasseattributter har navnet understreket (UML har ingen fast notasjon for klasseoperasjoner). Bruk gjerne restriksjonen {static}.
- ✓ Synligheten er angitt med + for public, - for private, # for protected, ~for package eller / for derived.
- ✓ Innenfor en klasse må alle operasjoner – inkludert de arvede – ha forskjellig signatur, dvs. operasjonsnavn + argumentenes type og rekkefølge (men ikke argumentenes navn, synlighet eller operasjonstype).
- ✓ UML *anbefaler* å skrive medlemsnavn med liten forbokstav og klassenavn med stor, men det er ikke obligatorisk..

Objekter

objektnavn : Klassenavn
attributtnavn = verdi

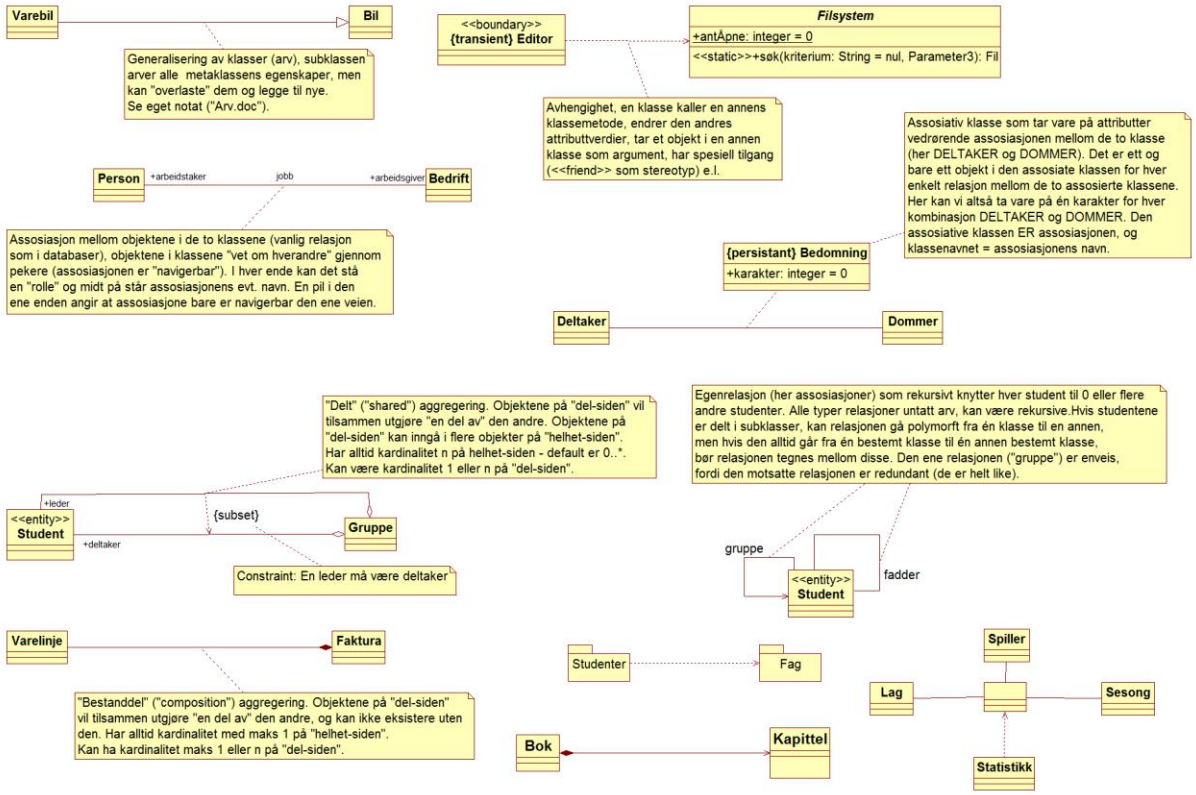
- ✓ Objekter har navnet understreket etterfulgt av kolon og klassens navn.
- ✓ Hvis objektnavnet ikke er oppgitt, er det et ”generisk objekt”, altså en eller annen instans av klassen. Da er vanligvis ikke attributtene angitt.

Synlighet

+	= public, dvs. synlig for alle som ha tilgang til objektet/klassen.
#	= protected, dvs. synlig for objektet/klassen selv og alle sub-objekter/sub-klasser som har tilgang til objektet/klassen.
-	= private, dvs. synlig bare for medlemmer i samme objekt/klasse.
~	= package, dvs. synlig for alle i samme pakke som har tilgang til objektet/klassen.
/	= derived, dvs. attributtet finnes ikke, og verdien lagres følgelig ikke, men verdien genereres (med tilgangsmetode) ved behov (ikke UML).

Note: I C++ kan klasser deklareres og deklareres med synlighet *friend* og er da bare synlig for angitt "venneklasser". UML har ikke denne synligheten – bruk evt. en restriksjon eller stereotype.

Relasjoner i klasse-/objektdiagrammer

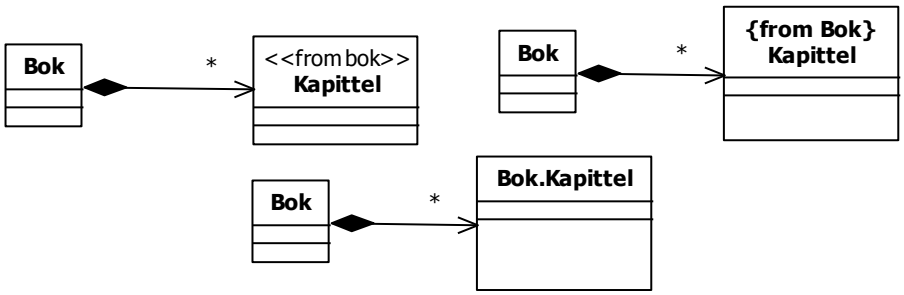


Kardinalitet

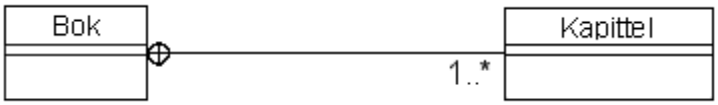
Med kardinalitet angitt med ett tall, f.eks. 5, menes akkurat tallet, f.eks. 5..5. Dette gjelder også for tallet 1. Unntak for * som pr definisjon betyr 0..*. Annet må angis. UML standard bruker * for ”mange”, men også n kan brukes. Kardinaliteten til aggregeringer er * for delt, og 1 for bestanddel, hvis ikke annet er angitt.

Nøstede klasser

Nøstede klasser brukes sjelden. Bare noen OOP-språk støtter det. UML støtter det ikke eksplisitt, men flere notasjoner/konvensjoner foreslås:

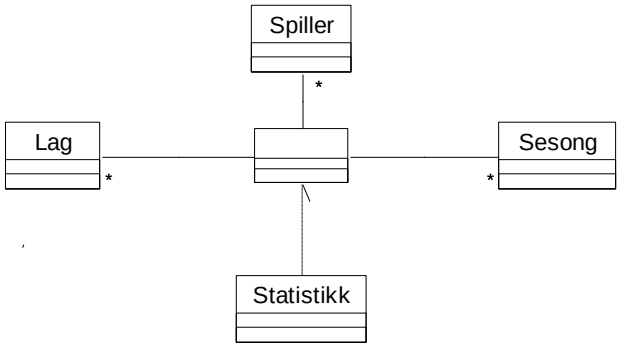


UML versjon 1.4 tegnet nøstede klasser ved å trekke en linje mellom klassene, med eget symbol for nøsting eller ”inneholder”:



Alternativt kan man jo bruke en pakke (namespace) *Bok* med klassene *Bok* og *Kapittel*, men meningen blir da ikke eksakt den samme.

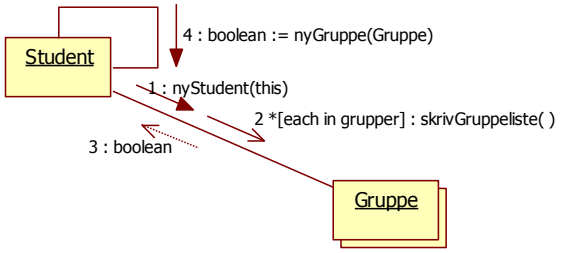
Treveis og flere (trinary og n-ary) relasjoner



Til hver kombinasjon av LAG og SPILLER er det knyttet flere SESONGer. Til hver kombinasjon av SPILLER og SESONG er det knyttet flere LAG osv. Til hver, enkelt forekomst av assosiasjon er det knyttet en assosiativ klasse STATISTIKK.

Sequence/Collaboration Diagrams (Sekvens-/samarbeidsdiagrammer)

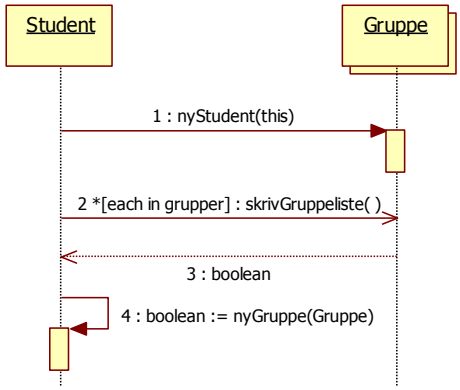
Samarbeidsdiagram:



Student er et ”anonymt” (generisk) objekt av Studentklassen. Det er vist en struktur med flere gruppeobjekter. *nyGruppe()* er et synkront, rekursivt kall. *nyStudent(this)* er en synkron melding, *skrivGruppeliste()* er asynkron og *boolean* er en returverdi. Meldinger forventes

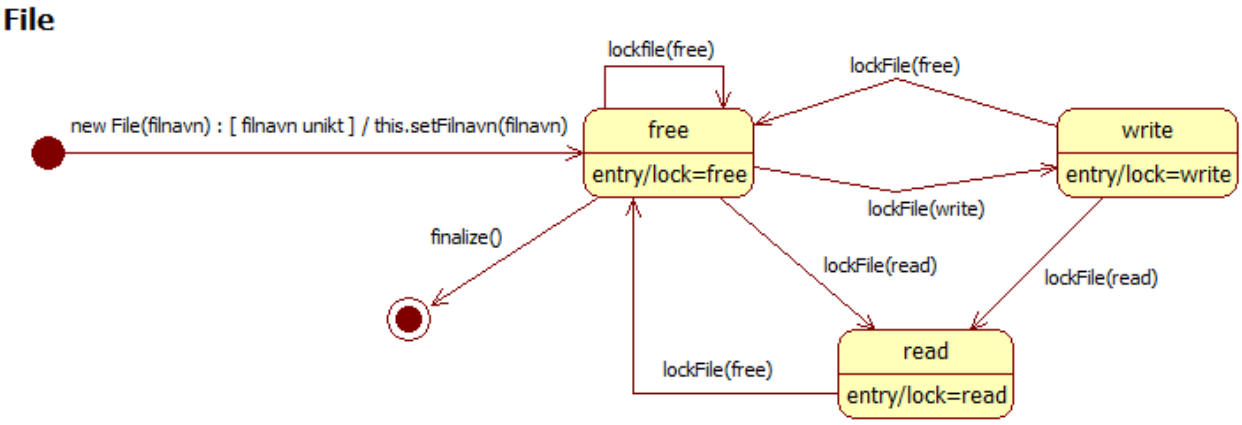
alltid å ha retur (i verste fall bare *void*). Det er valgfritt om man vil vise returverdien – alle meldinger har jo returverdi, men de er særlig aktuelle for asynkrone kall, og det blir ofte klarere om de angis spesielt For kallet *nyGruppe* er returverdien vist eksplisitt på en annen måte. For kallet *skrivGruppeliste()* er det også vist en iterasjon – her skal alle gruppeobjektene i strukturen *grupper* kalles (egentlig skal det stå et "while"-vilkår inne i hakeparentesene).

Det tilsvarende sekvensdiagrammet:



Det er tett sammenheng mellom sekvensdiagrammer og samarbeidsdiagrammer – de dokumenterer det samme med forskjellig fokus. Det ene kan lages automatisk fra det andre og det er en smaksak hvilket man vil starte med og hvilke man vil konvertere. Legg merke til at det synkrone kallet *nyStudent()* fører til at Student må vente til Gruppe er ferdig (vist som en loddrett søyle), mens det asynkrone kallet *skrivGruppeliste()* ikke gir tilsvarende venting. Det rekursive kallet *nyGruppe(Gruppe)* er synkront og objektet må "vente på seg selv".

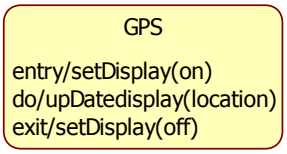
State Machine Diagram (Tilstandsdiagrammer)



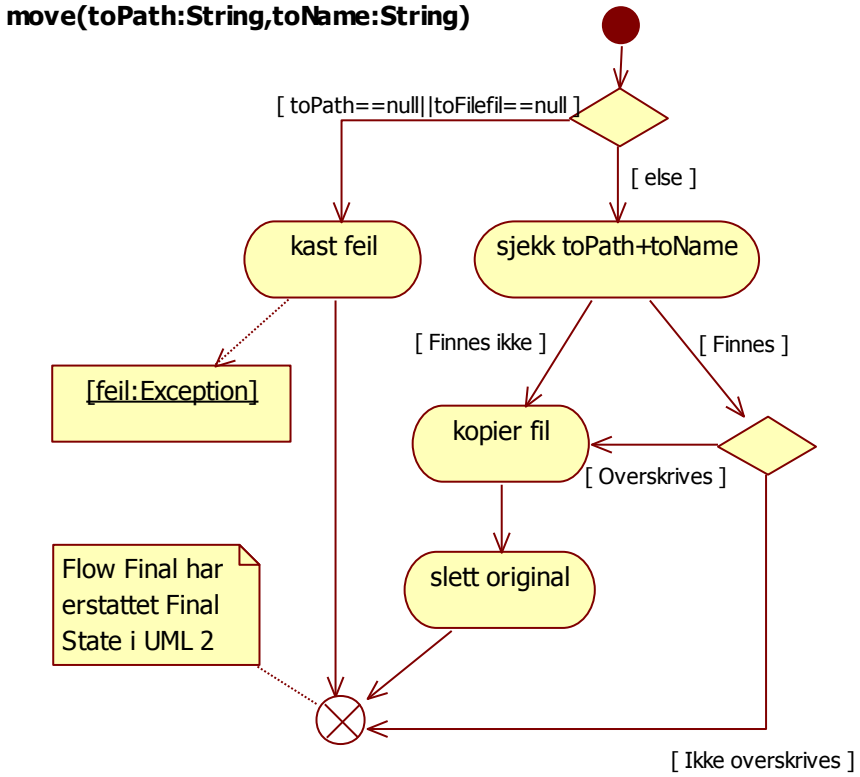
Relasjonene viser her tilstandsoverganger mellom tilstander. Overgangene kan være refleksive (til/fra samme tilstand) og rekursive (A til B til A). Relasjonene viser de stimuli som objektet skal reagere på i hver tilstand. Hvis en overgang ikke er angitt, skal objektet ikke reagere i det hele tatt. Som reaksjon kan objektet skifte tilstand og/eller gi respons i form av signaler eller meldinger til andre objekter. For hver tilstand kan man angi *entry action* = det som skal gjøres når tilstanden inntreffer, *do action* = handlinger som gjøres kontinuerlig mens objektet er i denne tilstanden (sjeldent) og *exit action* = gjøres når (rett før) tilstanden forlates.

Syntaksen for tilstandsovergangen er *trigger [skranke]/handling*. Triggeret er en hendelse som objektet oppdager, vanligvis en melding med aktuelle argumenter. Skranke er et vilkår

for at triggeret skal gi tilstandsovergang – det uttrykkes som et Boolsk uttrykk basert på objektets egne egenskaper og/eller parameterverdiene. Handlingene er bieffekter, f.eks. meldinger/signaler som sendes til andre objekter når triggeret mottas. Hvis tilstanden medfører samme handling for *alle* innkommende overganger, kan de bedre uttrykkes som en entry action og det inkluderer evt. returverdi (hvis meldingen er en funksjon).



Aktivitetsdiagrammer



Aktivitetsdiagrammer likner på ”gammeldagse” flytdiagrammer” og viser algoritmer, f.eks. algoritmen for en operasjon (metode). De er en form for tilstandsdiagrammer, men tilstandene er her *aktivitetstilstander*, dvs. aktiviteter som varer en stund (*action state*).

Det er mulig å vise *valg* enten med eget symbol eller som to utganger fra en tilstand med tilhørende *guard condition* som viser når det inntreffer. Det kan sendes ”objektstrøm” ut av metoden (til venstre er vist en feil) og mellom metoder (vist til høyre).

Figuren nedenfor viser dessuten et eksempel på parallelle løp som samles igjen. Dette diagrammet er også delt i ”swim lanes” som skal klargjøre diagrammet for leseren, men som ikke endrer ikke algoritmen på noen måte.

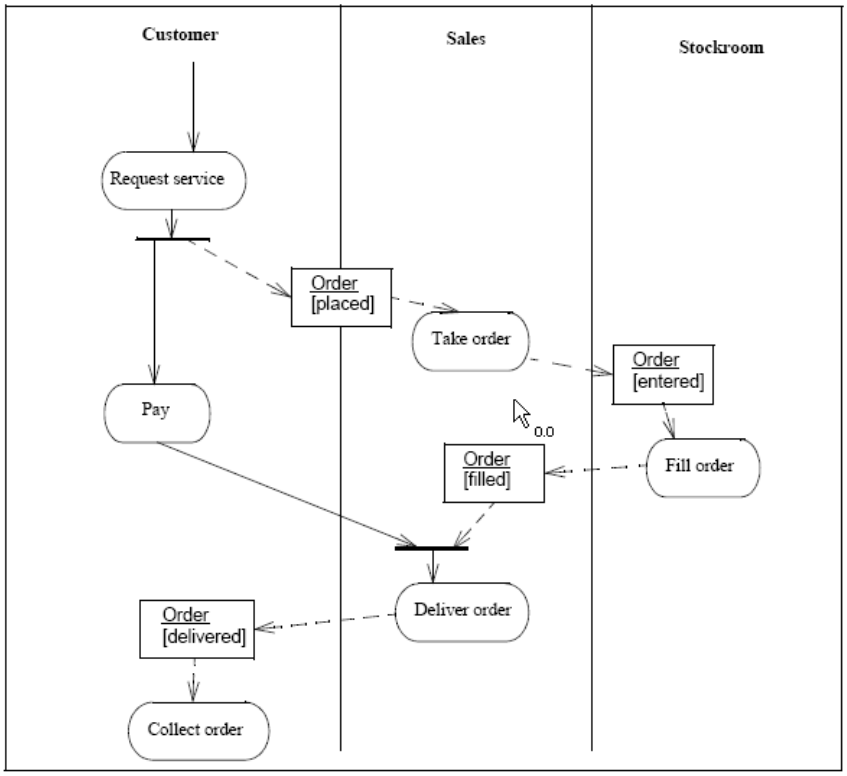
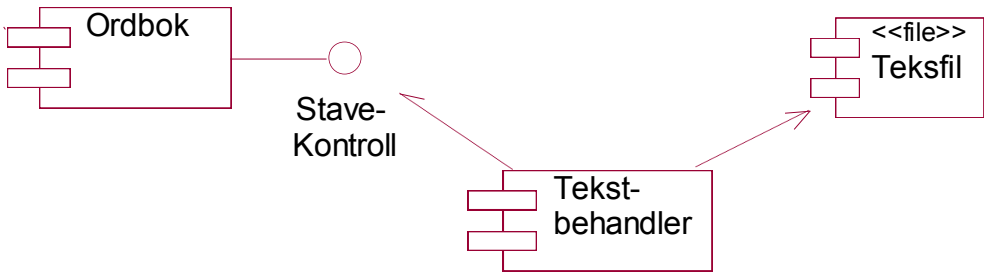


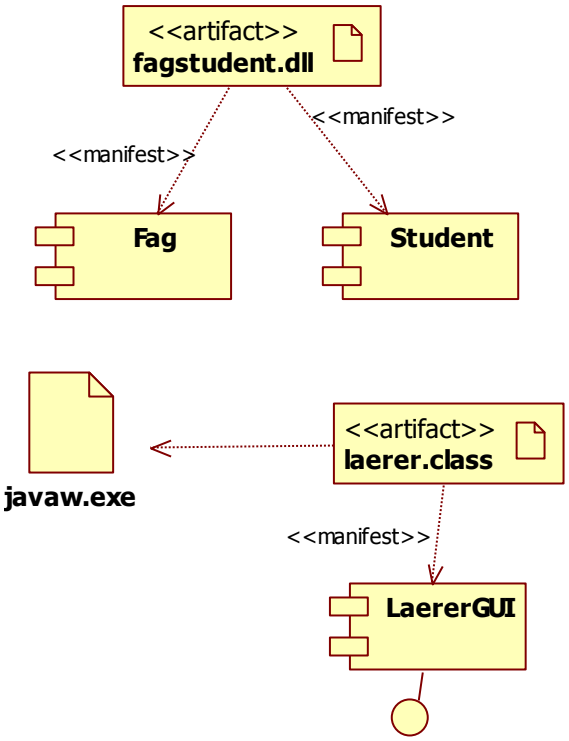
Figure 3-90 Actions and Object Flow

Component Diagrams (Komponentdiagrammer)



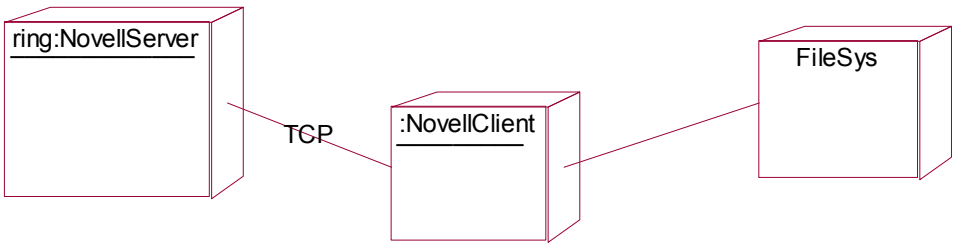
Komponenten *Tekstbehandler* er avhengig av (bruker) komponenten *Tekstfil* og komponenten *Ordbok* gjennom grensesnittet *StaveKontroll*. Grensesnittet utgjør en klasse, og kan også tegnes som det, men det er uvanlig. Det vil være mer aktuelt å spesifisere grensesnittet med CORBA IDL.

Komponentene samles i ***artefakter***, som er fysiske representasjoner av én/flere komponenter:



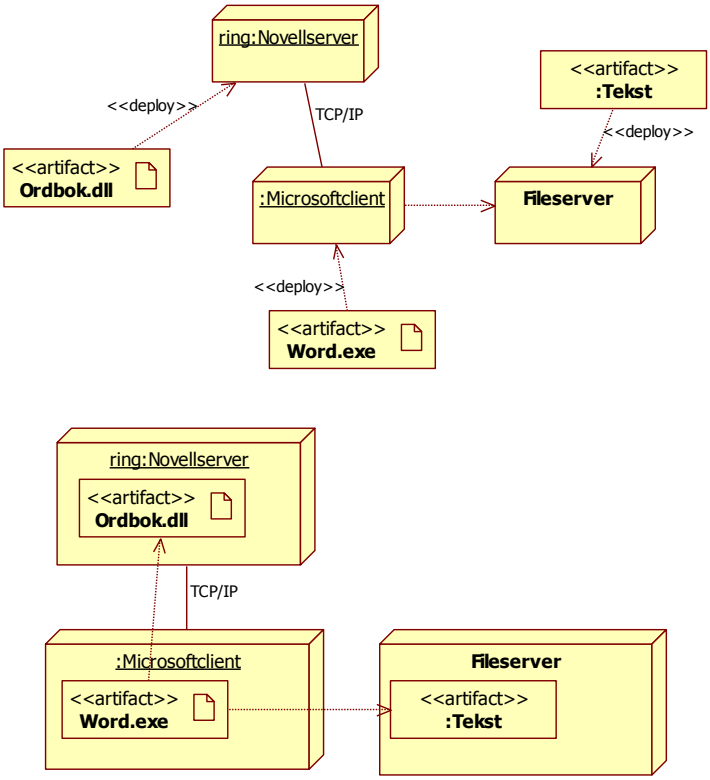
Fag- og *studentkomponentene* samles i artefaktet *fagstudent.dll* som er et programbibliotek. *laerer.class* er en Java-klasse som manifesterer komponenten *LaererGUI* og den er avhengig av artefaktet *javaw.exe* (Java runtime for Microsoft OS). *javaw.exe* viser alternativ tegnemåte.

Deployment Diagrams (Utplasserings- eller Nodediagrammer)



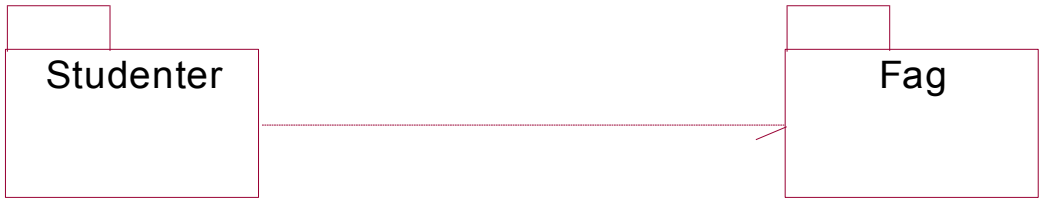
ring:NovellServer er en navngitt instans av nodeklassen *NovellServer*. *:NovellClient* er en anonym (=”en eller annen”) instans av nodeklassen NovellClient, mens *FileSys* er en nodeklasse.

Nodene vil inneholde kopier av artefaktene under kjøring – artefaktene *utplasseres* (<<deploy>>) i nodene:



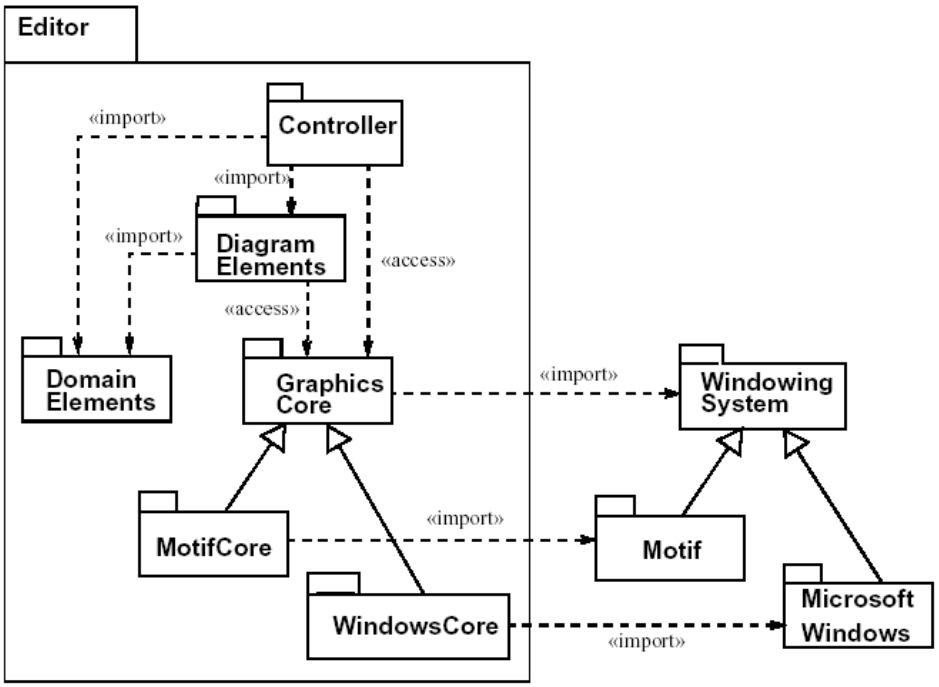
Det er her vist to – av flere – notasjoner. Artefaktet *Ordbok.dll* utplasseres i noden *ring:Novellserver*. Den er koblet med TCP/IP til en anonym node *MicrosoftClient* som har utplasser artefaktet *Word.exe*. Denne noden er igjen er avhengig av noden *Fileserver* med artefaktet *:Tekst*. Det er anledning til å benytte andre stereotyper som er mer presise en <<artefact>>, f.eks. <<document>>, <<executable>>, <<file>>, <<library>> eller <<source>>. Her vil vel da *Ordbok.dll* og *Word.exe* være <<executable>>, mens *Tekst* er en <<file>>.

Packages (pakker) – kan brukes i alle diagrammer

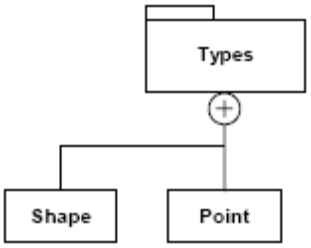


Her er pakken *Studenter* er avhengig av pakken fag, dvs at noe av innholdet i pakken *Studenter* har en relasjon til noe av innholdet i pakken *Fag*.

Pakker er bare en måte å gruppere elementer i modellen på, og har ingen annen mening hverken logisk eller fysisk.. Det kan sammenliknes med ”navnerom”. Det er tillatt å vise innholdet i pakken, og da skal navnet stå på ”fanen”:



Alternativt kan innholdet vises med ”nøsting”:



Høgskolen i Buskerud
Postboks 235
3603 Kongsberg
Telefon: 32 86 95 00

www.hibu.no

ISSN 1893-2398 (online)

HØGSKOLEN
i Buskerud