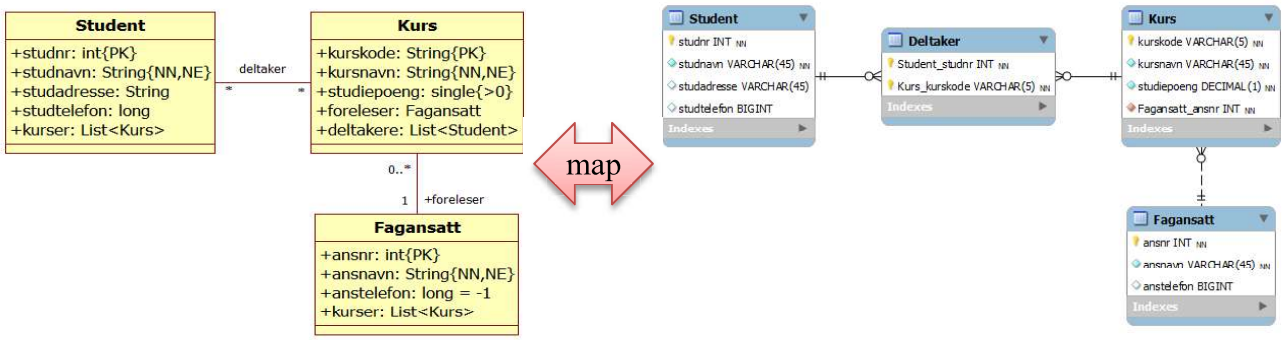
Kapittel 8 – Persistens av objekter: OOP & RDBMS

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.

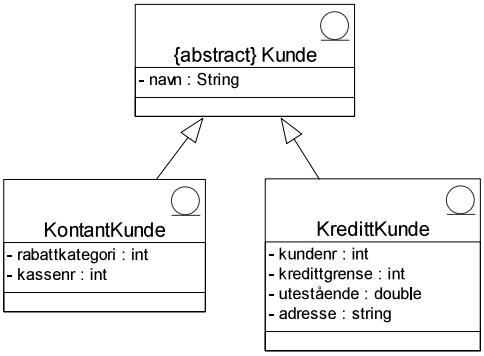
[34] I *OODBMS* regnes de ofte forskjellige bare de har minst én attributtverdi forskjellig, men det kan *variere* hvilket attributt det er. Slike databaser kan ofte også selv, automatisk legge til et eget ID-attributt, kalt Object Identifier, forkortet OID.

[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:

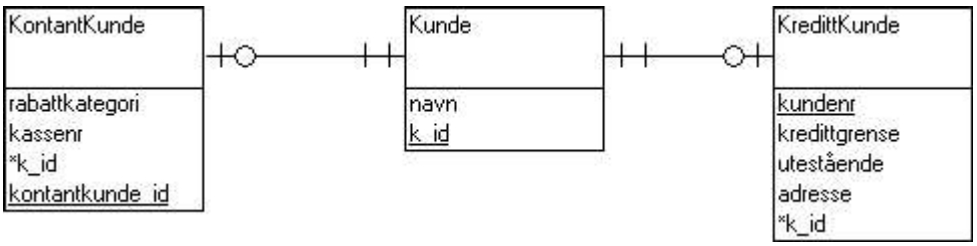


A) Vertikal mapping (separasjon)

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)

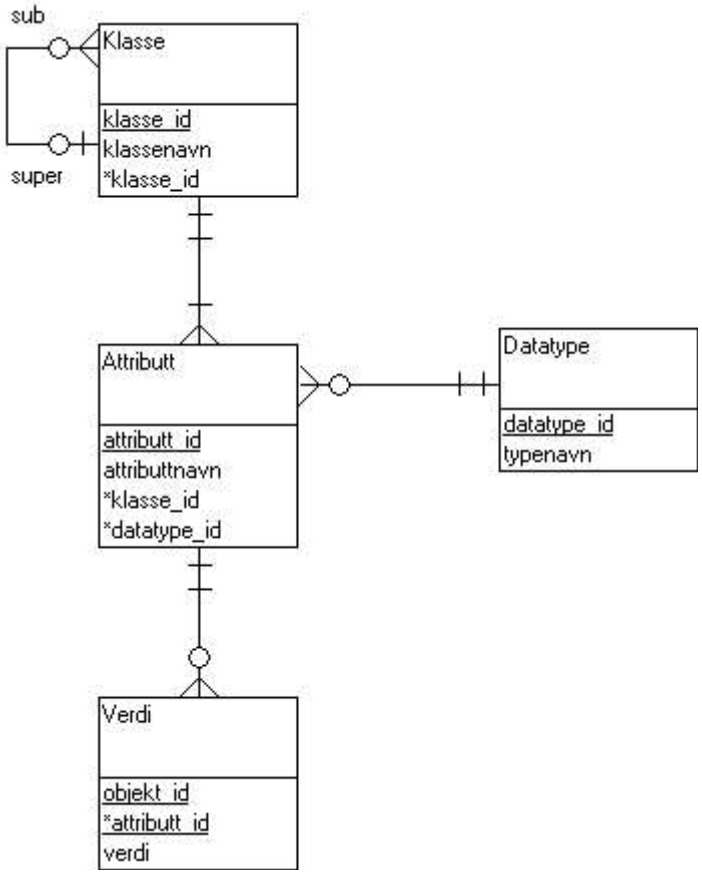
Alle klassene i et hierarki slås sammen til én tabell. I tillegg kreves da en attributt som angir hvilken klasse posten tilhører, og selvsagt en PK.

```
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.

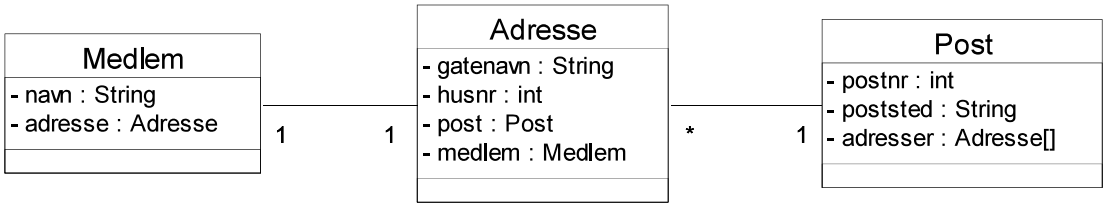


Mapping av noen spesielle situasjoner

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

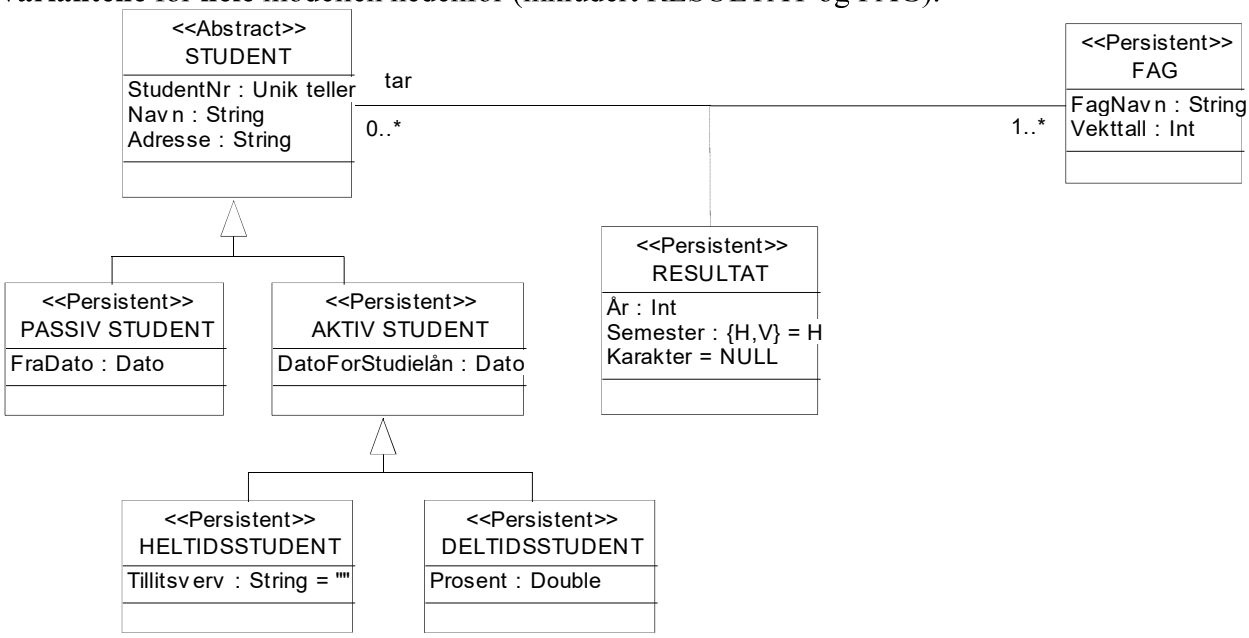
Aggregeringer (”består av”) krever triggere og/eller kaskadeeffekter, siden delene må slettes når aggregeringen slettes.

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.