

Objektorientert programmering og systemutvikling - en kortfattet innføring

Arne Maus
Institutt for informatikk
Univ. i Oslo

11. september 2002

Dette notatet er ment å være en oppsummering og utfyllende notat av det stoffet som gjennomgås i de generelle delene i kurset i “ Objektorientert systemutvikling”, men kan også leses som en frittstående innføring i emnet. Notatet inneholder eksempler skissert i C++ , men er ikke på ingen måte en tilstrekkelig innføring i C++. Vekten er lagt på en forklaring av begrepene og de generelle språkmekanismene som nyttes i objektorientert analyse, design og programmering.

1 Innledning

Store programmer er vanskelige å lage, feilsøke og vedlikeholde. Allerede tidlig på 50-tallet ble man klar over at man trengte hjelpemidler innen programmering, og vi fikk utviklet ulike typer av programmeringsspråk som et svar på dette behovet. Med disse språkene fikk vi innført flere nye begreper og teknikker som fremdeles nyttes. Utviklingen på 50-tallet peker framover mot utviklingen av objektorientert programmering på 60-tallet.

Først kom assemblerne som grovt sett hjalp til å få et skille mellom program og data, samt at selve handlingssekvensene ble mer leselige. Isteden for binære tallverdier som navn på operasjoner, fikk vi navn på operasjonene som ADD og MULT. Selv om det hevet produktiviteten sterkt, viste det seg snart at man kunne forbedre situasjonen ytterligere.

Neste forsøk på å lage hjelpemidler for programmering tok sitt utspring i selve handlingssekvensen. Med FORTRAN og Algol fikk vi programmeringsspråk i som med et funksjons- og prosedyrebegrep gjør det mulig å pakke en sekvens av enkelthandlinger sammen i en enhet og gi denne et navn som beskriver denne sammensatte handlingen.

En slik prosedyre kunne også gjøres mer generell ved å gi den parametre. En prosedyre representerer en generell løsning på et mindre og enklere problem enn det problemet som hele programmet skal løse. Vi har på denne måte delt inn handlingene i programmet inn i mindre grupper av handlinger og vi har gitt hver av disse navn. Siden mange problemer har samme typer av subproblemer, blir det også på denne måten mye lettere å gjenbruke kode ved at slike prosedyrer eller subrutiner kan samles i subrutinebiblioteker.

Et annet forsøk på å få mer orden og oversikt i programmene kom fra administrativ databehandling med Cobol. Særlig for å få mer orden på innholdet av filer på disk og magnetbånd, men også innad i programmene, fant man det naturlig å slå sammen de variable som logisk hørte sammen i records med navn. Dels kan man operere på en record som en enhet (f.eks lese eller skrive den til

disk), og dels kan man få adgang til de enkelte delene av denne ved å adressere enkeltelementene.

Fra databasehold har man også arbeidet videre med denne struktureringen av datasystemet rundt en fokusering på data, ved å tilby ulike verktøy og begreper for å lage en sammenhengende data-modell for hele problemet vi skal løse. Både de prosedyreorienterte programmerings-språkene og dataorientert programmering, f.eks i form av Cobol, lever idag i beste velgående, og utgjør klart hoveddelen av dagens programmering. De teknikkene som der ble utviklet, er grunnelementer i de fleste moderne programmeringsspråk.

Ikke desto mindre følte man at noe manglet, at kanskje mange av de problemene man opplevde i disse to programmeringsstilene kunne avhjelpes ved en mer helhetlig og ballansert holding til både prosedyrer og datamodeller.

2 Objektorientert programmering

Hovedideen ved objektorientert programmering består i at man innser at det til de data (den record) som beskriver en "gjenstand" i vårt system tilhører et visst antall operasjoner som naturlig opererer på disse data. Disse operasjonene kan programmeres som prosedyrer og ved hjelp av mekanismer i programmeringsspråket kan de bli koblet sammen med tilhørende databeskrivelse (record).

Det første språket som tilbød slike muligheter var Simula som ble utviklet ved Norsk Regnesentral på 60-tallet. Det var ikke helt uventet at dette språket kom fra et programmeringsmiljø som laget simuleringssystemer.

Når vi programmerer, kan vi si at vi lager en modell av den delen av virkeligheten vi er interessert i. Vi lager da i en viss forstand alltid en simulering av virkeligheten og at vårt program modellerer eller simulerer denne delen av virkeligheten. I objektorientert programmering legger man spesiell vekt på dette aspektet.

2.1 Klasser og objekter

En slik deklarasjon av en datarecord med tilhørende prosedyrer kalles en klasse. Når vi under eksekvering av programmet oppretter et eksemplar av en slik klasse, kalles det et objekt. Grunnen til å skille mellom deklarasjonen og et eksemplar av en slik deklarasjon under eksekvering, er at det kan tenkes at vi lager mange eksemplarer av samme deklarasjon, og at disse objektene kan eksistere samtidig under utførelsen av programmet.

Dette skillet mellom selve deklarasjonen (klassen) og hvert eksemplar (objektene) kan eksemplifiseres ved å tenke oss at vi lager et program som simulerer biltrafikken på Mosseveien. Vi vil da ha bare en deklarasjon av hvordan en bil skal representeres (bil-klassen), men vi må ha svært mange bil-objekter samtidig for å få til en realistisk simulering av f.eks kødannelsene.

Når vi lager en slik objektorientert programløsning, identifiserer vi først hvilke "gjenstander" eller begreper i vår modell som skal modelleres med objekter. Deretter fastsetter vi hvilke egenskaper ved disse objektene vi er interessert i og hvilke operasjoner vi ønsker å gjøre på disse operasjonene. Ut fra denne litt abstrakte systemeringen av objektene foretar vi så den konkrete implementasjonen ved å deklare den datarecord vi skal ha og skriver en prosedyre for hver operasjon. Legg merke til at etter vi har bestemt hvilke 'abstrakte' data et objekt har, er det ikke alltid opplagt hvordan dette er representert i objektets datarecord. F.eks kan navnet til eieren av en bil i klassen 'bil' være representert ved et tekstfelt - alternativt ved at vi slår opp eierens navn i et bil-register ved hjelp av bilnummeret.

Et par eksempler vil illustrere dette forholdet at data og operasjonene på disse tilsammen utgjør en naturlig enhet - en klasse. Anta at vi ønsker å lage et eksamensregister for studenter. Datadelen av klassen som beskriver en student, vil da typisk inneholde felter for navn, adresse, personnummer samt et visst antall plasser for samlinger av kursnavn, eksamensdato og karakter. Naturlige operasjoner for denne klassen kunne da være:

- `meld_opp_til_eksamen(kurs_nummer)`
- `registrer_eksamenskarakter()`
- `regn_ut_gjennnitt_karakter()`
- `skriv_ut_vitnemål()`

Disse fire operasjonene (sammen med flere andre operasjoner i et virkelig system) utgjør da for vårt formål de interessante operasjonene 'på' en student. Legg merke til at hvilken student det dreier seg om i disse prosedyrene ikke omtales (ikke er parameter) fordi dette selvsagt er den studenten som data-delen av det aktuelle objektet identifiserer. Sammen med datadelen utgjør deklarasjonen av disse fire prosedyrene student-klassen. For å identifisere hvert enkelt objekt (en bestemt student) brukes pekervariable, som blir forklart litt senere.

Et annet eksempel kan være et banksystem hvor vi tenker oss at hver konto er representert av et objekt av klassen 'konto'. Denne vil da ha deklareret datafelter som saldo, liste over posteringer, kontonummer, eierens navn, adresse,..osv. Operasjoner vil kunne være prosedyrer som 'sett_inn(beløp)', 'ta_ut(beløp)', 'beregner_rente()', osv.

Av disse to eksemplene kan vi se to ting. Det første er at mens vi omlag kan nytte samme data-record til å beskrive en student eller en konto som vi ville i et Cobol-program, vil ikke prosedyrene nødvendigvis tilsvare de man ville finne i en Fortran, Algol, Pascal eller C-løsning. Vi må nå skrive prosedyrene slik at de passer til å være en operasjon til data for ett objekt (i våre eksempler for en student eller en konto).

Det andre vi observerer er at det godt kan være flere objekter av samme type samtidig tilstede i systemet. Vi må tilsvarende som i bileksempelen kunne ha flere studenter samtidig i eksamensregisteret og flere konti i bankeksempelet.

For å oppsummere, et objekt består av to deler: a) en datadel som beskriver egenskapene til den "gjenstanden" vi modellerer, og b) tilhørende operasjoner på disse data.

I et objektorientert program vil vi også ha et vanlig hovedprogram og noen vanlige prosedyrer (ikke tilknyttet noen klasse) i tillegg til klassene. De mer vanlige prosedyrene vil ofte være slike som naturlig opererer på flere klasser samtidig, mens man i hovedprogrammet typisk vil starte opp programmet med å opprette noen objekter. Hoveddelen av handlingene og dataene vil imidlertid ligge i klassene, og handlingene i programmet vil i stor grad skje ved at man fra prosedyrer i objektene kaller prosedyrer i andre objekter.

2.2 Pekervariable og opprettelse av objekter

Når vi nå kan lage store objekter, og gjerne mange av disse samtidig, oppstår et behov for en ny type variabel: Pekervariabelen. Pekervariable brukes til å peke på objekter av en klasse (på engelsk: ref - variable). Dette er en variabel som egentlig inneholder adressen i lageret til hvor objektet som pekes på, ligger. I mange språk som C++ og Simula, er det slik at en pekervariable kan bare peke på

objekter av en gitt klasse og man må si fra hvilken klasse dette er når referansevariabelen deklarerer (se fig 1).

Tar vi utgangspunktet i bilkø eksempelet vårt, og antar at vi er interessert i å representere en bilkø ved f.eks bomringen, kunne vi trenge en pekervariabel i hovedprogrammet som pekte på første bil i denne køen. Hvis så klassen bil inneholdt en pekervariabel som peker på neste bil i køen, kunne vi lett representere bilkøen ved at første bilobjekts pekervariabel pekte på bil nr.2 i køen,..osv. Peger i siste bilobjekt i køen ville da ikke ha noen bil å peke på og ville inneholde verdien 'null'.

```
class bil
{
    bil* neste;
    int reg_nummer;
    .....
};

bil* første;
.....
første = new bil;
første -> reg_nummer = 443396;
.....
første -> neste = new bil;
/* Kommentar: Det er nå 2 biler i køen */
.....
```

Programeksempel 1

Programeksempel 1, som er skrevet i C++, viser både hvordan man deklarerer pekervariabler til objekter av klassen bil (**bil***) og hvordan man oppretter nye objekter av en bestemt klasse (ved **new** - operatoren).

Det er ikke mange meningsfulle operasjoner vi kan gjøre på innholdet av en pekervariabel. Vi kan la den peke på et nyopprettet objekt som i eksempelet ovenfor og vi kan la den peke på et annet objekt ved at innholdet av en pekervariabel (som peker på dette andre objektet) kopieres over i den første pekervariabelen.

Pekervariabler har imidlertid en helt vesentlig funksjon i tillegg til å peke på ett bestemt objekt. Det er via en pekervariabel vi kan få adgang til innholdet av objekter, dvs. til de enkelte datafeltene og prosedyrene. I eksempelet ovenfor settes det en verdi inn i registreringsnummeret til den første bilen med uttrykket: 'første->reg_nummer = 443396;' (Vi kan lese dette som 'første sitt reg_nummer' settes lik ...).

Dette uttrykket 'første->reg_nummer' kan brukes på helt samme måte som vi kan bruke navnet på enhver annen deklartert heltallsvariabel, f.eks både på høyre og venstre side av en tilordningssetning. Et slik eksempel kan være:

```
første->reg_nummer = første->reg_nummer + 3;
```

Bort sett fra at det er logisk sett en meningsløs operasjon å legge verdien 3 til et bilnummer, viser den hvordan det er mulig å få adgang til innholdet av et objekt ved hjelp av en pekervariabel.

Prosedyrer i et objekt får man adgang til på helt samme måte som til enkle variable. Anta at klassen bil har en prosedyre som returnerer et heltall: 'antall_kjorte_km()'. Man vil da få kalt opp denne

prosedyren i det objektet som pekes på av 'første'-pekervariabelen ved uttrykket: 'første->antall_kjørte_km()'. Dette kan vi gjøre ethvert sted i programteksten hvor vi (på vanlig) måte har adgang til pekervariabelen 'første'. Dette eksemplet viser den måten vi får adgang til egenskapene til et objekt - man har en pekervariabel som peker på objektet, og så bruker vi navnet på denne variabelen og en spesiell operator (i C++: ->) til å få adgang til "innmaten" av objektet.

Tilsvarende som prosedyrer kan også klasser ha parametre. Når vi oppretter et objekt av en klasse, kan vi som aktuelle parametre sende med vanlige variable som heltall, arrayer, tekster,..osv samt variable som er pekervariable. De nye objektet kan da som parameter få en (eller flere pekere) som parameter og dermed få adgang til andre objekter (av samme klasse som seg selv eller av andre klasser). Det er meget vanlig at objektene på denne måten "peker på" hverandre.

-- o o --

Det overstående beskriver vesentlige aspekter ved objektorientert programmering: En objektorientert løsning består i all hovedsak av en eller flere objekter fra en eller flere klasser. Disse objektene holdes sammen med, og innholdet får man adgang til, via pekervariable. Det er imidlertid flere interessante spørsmål og to nye mekanismer ved objektorientert systemering og programmering som drøftes i de neste avsnitt.

3 Hva er objekter, systemering

Når vi lager et system, velger vi som før nevnt ut bare den del av virkeligheten som vi er interessert i for vårt formål og vi representerer bare de data og handlinger (prosedyrer) som blir viktige i forhold til dette. Det er imidlertid ikke alltid opplagt hva vi bør velge ut som objekter i vårt system.

Tar vi eksempelet med studenteksamensregisteret, kan vi kanskje bare klare oss med studentobjekter, men kanskje vil vi også representere de ulike kursene og evt også studieretningene som objekter. Alt kommer an på hvilken vinkling systemet har, hvilke spørsmål vi er interessert i å få svar på.

Et annet eksempel på hvordan objekter kan være valgt, er at i et moderne grafisk brukergrensesnitt med vinduer, menyer, rullefelt og tekstbokser på skjermen. Hver av de feltene som forekommer på skjermen vil typisk i programmet være representert med hvert sitt objekt i det programmet som administrerer vinduene.

En måte å gjøre det på er da å la det være deklartert en klasse for hver type av objekt vi kan ha på skjermen (menyer, rullefelter, tekstbokser,...). Trenger vi så f.eks tre tekstbokser samtidig på skjermen, oppretter vi så tre objekter av tekstboks-klassen. De operasjonene som typisk vil være tilknyttet og deklartert inne i et tekstboks-klassen, vil være operasjoner som 'les teksten i boksen', 'skriv tekst i boksen', 'skjul tekstboksen med innhold', 'vis fram',.. osv.

I mer kompliserte klasser som rullefelter vil vi kunne ha deklartert operasjoner som legger inn en ny tekstlinje, teller antall tekstlinjer i rullefeltet, en prosedyre som blir kalt hvis brukeren med musa klikker på en linje i rullefeltet (med returverdi på hvilken tekstlinje som ble valgt), fjerner en bestemt linje,..osv. I et virkelig vindussystem som f.eks Windows95/NT vil det i sum være godt over 500 slike operasjoner .

3.1 Objektters indre og ytre egenskaper - aksessprosedyrer

Vi har i et tidligere avsnitt sett at det ikke alltid er opplagt hva vi skal velge som klasser. Det er ingen enkel, naturgitt oppskrift som forteller oss dette. Valg av objekter og spesifisering av hva som er objekter og hva disse skal ha som egenskaper er like vanskelig som enhver annen systemering. Noen generelle retningslinjer kan imidlertid gis.

For det første vil vi ofte lage klasser til de vesentlige fysiske objekter eller hovedbegreper i den delen av virkeligheten vi lager et datasystem for. Antar vi at vi lager et billettreservasjonssystem for charterreiser, vil vi kanskje lage en objekttype (klasse) for reisemål, for fly, for kunder og for hoteller. Kanskje er en billettreservasjon også en klasse eller kanskje er denne informasjonen innholdt i kunde-klassen.

Et annet og viktig kriterium ved valg av objekter angår en erfaring vi også har fra annen programmering: Noen variable og operasjoner (prosedyrer) kan betraktes som hjelpevariable og hjelpeoperasjoner, mens andre variable og operasjoner representerer det "egentlige" problemet. Dette skillet blir eksplisitt utnyttet i objektorientert programmering. Vi kan ved hjelp av mekanismer i programmeringsspråk som C++ effektivt dele variable og prosedyrer i en klasse i to:

- de ytre, essensielle data som beskriver den gjenstanden som klassen representerer og de logiske operasjonene vi kan gjøre på disse data. I vårt studentregister-eksempel er dette mellom annet data som navn og karakterer samt operasjoner som 'registrer_en_ny_eksamen()' og 'skriv_ut_vitnemål()'. Disse prosedyrene, som definerer de interessante operasjoner på de "ytre data", kalles ofte aksessprosedyrer.
- de indre hjelpevariable og hjelpeoperasjonene som har å gjøre med selve implementasjonen. Dette kan være alt fra enkle tellevariable i løkker til kompliserte datastrukturer som bygges opp for f.eks å få til en spesielt effektiv implementasjon.

En av de viktigste fordelene ved objektorientert programmering er at mens de ytre/logiske data og operasjoner er tilgjengelig fra de andre klassene og hovedprogrammet (via pekervariable), så gjør et objektorientert språk som C++ det lett å skjule disse indre hjelpevariable og hjelpeoperasjonene. Vi skjuler da hvordan vi har implementert de ytre egenskapene til objektet. Fordelene med dette er både at vi får delt opp programmet i flere, nesten uavhengige delmoduler, og at selve implementasjonen kan endres hvis det viser seg ønskelig uten at selve definisjonen og bruken av av klassen fra resten av programmet endres. Vi kan på denne måte endre måten et delproblem i form av en klasse er løst på, uten at det får ringvirkninger.

Det regnes som "god objektorientert programmeringsstil" ikke å lese og i alle fall ikke endre (skrive) direkte i et objekts variable, men alltid skrive aksessprosedyrer i klassene som foretar de nødvendige operasjonene. En hovedgrunn til en slik programmeringsstil er at endringer/skriving som oftest ikke er en enkel operasjon på en enkelt variabel, men en sammensatt operasjon på flere variable. Hvis vi lar andre klasser eller hovedprogrammet gå direkte inn i et objekts variable, fjerner vi samtidig våre muligheter til lett å kunne endre implementasjonen av objektets operasjoner uten å måtte undersøke alle steder hvor objekter av denne klassen brukes. Hvis all bruk av objektet går via aksessprosedyrer, har vi lett adgang til å skifte implementasjonen til objektet uten at dette får ringvirkninger i det øvrige programmet. En slik bruk av klasser og objekter implementerer det mange kaller: 'abstrakte datatyper'.

Begrepet 'aksessprosedyrer' gir et annet viktig prinsipp for hvordan vi skal velge våre klasser. Vi skal ha en slik oppdeling i klasser som gir færrest og enklest mulige aksessprosedyrer i klassene og slik at mulige endringer i programmet består i naturlige tillegg til de eksisterende klassene med

nye operasjoner og datafelter, evt. som en utvidelse med helt nye klasser. Mulige endringer bør ikke medføre at vi helt må kaste om på definisjonen av våre gamle klasser.

En god programmerer skal altså ikke bare lage en god struktur for den foreliggende spesifikasjon. Hun eller han skal også kunne tenke seg sannsynlige forandringer/utvidelser i spesifikasjonene, og lage strukturer i programmet slik at disse nye, utvidete spesifikasjonene blir lette å implementere.

3.2 Store eller små objekter

Et annet vesentlig spørsmål, er hvor store klasser vi bør definere. Bør vi ha mange små eller noen få store klasser. Noen objektorienterte språk som Smalltalk tar den konsekvente holdning at alt er objekter, også så små enheter som vanlige heltallsvariable (aksessprosedyrene til et slikt heltallsobjekt blir da operasjoner som +, -, *,...osv). I praksis vil dette gi så mange objekter at det kan være vanskelig å få oversikt.

I mer vanlige objektorienterte språk som C++, Simula, Eiffel, Java og Beta vil deklarasjonen av klassene typisk bestå av fra noen titalls til endel hundre linjer kode. Mens hver prosedyre ofte blir endel mindre enn det vi er vant til fra f.eks et Fortran-bibliotek, så er nok en klasse med alle sine data og prosedyrer i sum lengere enn en slik Fortran-prosedyre.

Av effektivitetsgrunner er det få grunner til å være sparsom med klassedeklarasjonene eller hvor mange objekter vi lager av hver klasse. Det er omlag like raskt å opprette et nytt objekt som å kalle en prosedyre i et vanlig programmeringsspråk. Et nytt problem har vi riktignok i objektorientert programmering. Når vi har objekter som vi ikke trenger lenger bør de leveres tilbake slik at plassen de opptok kan benyttes til andre objekter.

I noen objektorienterte språk som C++ må programmereren selv sørge for å gi tilbake "brukte" objekter til lageradministratoren, mens i språk som Simula og Java ordnes det automatisk av runtime-systemet, ved at når tilgjengelig plass nesten er oppbrukt, går systemet gjennom hele hukommelsen og fjerner objekter som ikke lenger kan nås via pekervariable (og som følgelig aldri mer kan brukes). En slik opprydding kalles søppeltømming (engelsk: garbage collection). Dette er noe enklere for programmereren, men den kan til gjengjeld gi et noe rykkvis oppførsel av programmet under eksekvering, da en slik søppeltømming kan ta flere sekunder.

Som en oppsummering kan vi si at vi velger objekter på en naturlig måte i forhold til fysiske objekter eller sentrale begreper i den delen av virkeligheten som programmet skal modellere. Dette må gjøres slik at alle objekter har så få og klart definerte aksessprosedyrer som mulig og at hver klasse ikke er større enn at en programmerer relativt lett kan få oversikt over koden for mulig feilretting, endring eller utvidelse. Etter at vi har introdusert noen flere begreper og språkmekanismer i neste kapittel, vil vi søke å gi en mer punktvis metode for objektorientert analyse og design i kap. 5.

4 Mer avanserte emner

Med et håp at overskriften ikke skremmer, skal vi i dette kapittelet se på to vesentlige tillegg til de objektorienterte mekanismene som hittil er presentert. Disse mekanismene, arv av egenskaper og redefinering av operasjoner, kan ofte nyttes til å lage langt mer elegante objektorienterte løsninger .

4.1 Arv av egenskaper - subclasser

Hvis vi lager et programsystem ved hjelp av klasser, vil vi av og til oppdage at vi er nødt til å definere helt tilsvarende operasjoner og/eller datastrukturer i flere av klassene. Dette kommer selvsagt av at flere av de “ting” i virkeligheten som vi modellerer med programmet, har mange felles trekk.

I vårt bankeeksempel hadde vi en klasse som beskrev en konto, og flere objekter av denne. I et virkelig system vil vi oppdage at det er mange typer av konti (honnørkonto, høyrentekonto, Boligsparkonto, lønnskonto,..osv). Selv om disse hver for seg har spesielle egenskaper, så har de også mange fellestrekk.

```
class konto
{ char[100] navn, adresse;
  int saldo, rentefot, kontonummer;
  .....
};
```

```
class lønnskonto: public konto
{ char[100] arbeidsgiver;
  int arbeidsgiver_kontonummer;
  .....
};
```

```
class Boligspar: public konto
{ int årlig_sparebeløp;
  .....
};
.....
```

(Her er prosedyrene ikke vist fram for å få uttrykt hovedsaken)

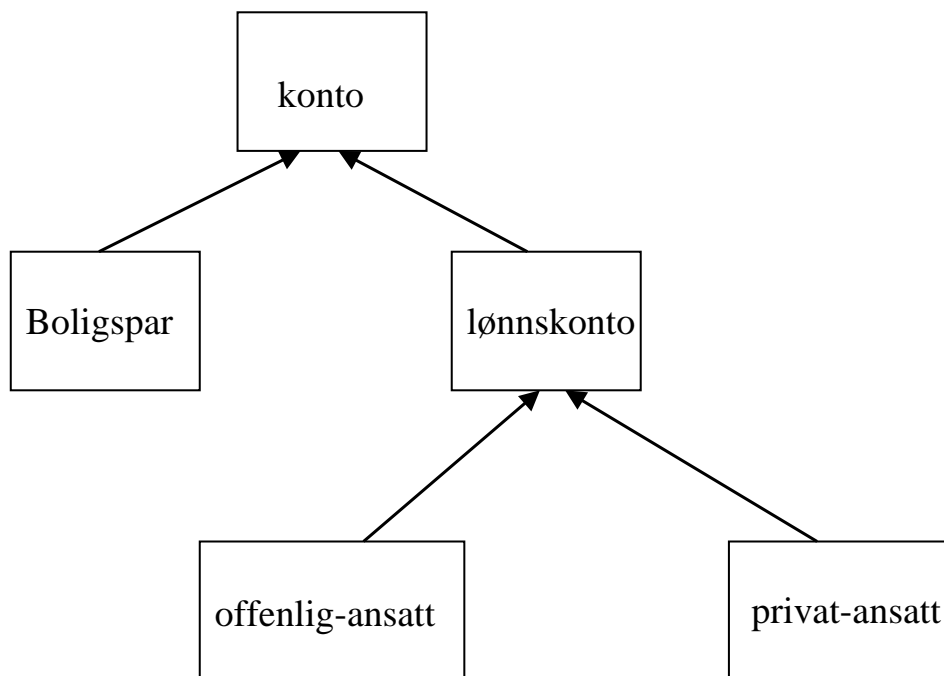
Programeksempel 2

En måte å løse dette problemet med fellestrekk og spesielle egenskaper **som ikke anbefales** er å deklare en eneste klasse som inneholder alle muligheter i form av datafelter og operasjoner, og så via statusvariable og parametre få fram spesialiseringen . Dette vil gi alt for mange datafelter i hvert objekt og rotete og uoversiktlig kode.

Det som anbefales, og som vi kan få til med objektorienterte språk, er at vi først skriver en klasse “konto” som inneholder alle felles data og felles operasjoner for alle kontotyper. For hver av de spesielle kontotypene kan vi så bare skrive de tillegg og spesialtilfeller som trengs for hver enkelt kontotype. Vi kan så skjøte hver av disse tilleggsdeklarasjonene sammen med klassen konto og gjør dermed disse tilleggsklassene til såkalte subclasser til klassen 'konto'. Har vi f.eks skrevet de tilleggene som trengs for en lønnskotoklasse og en Boligspar klasse, kan vi bare ved å nevne konto-klassenavnet sammen med deklarasjonene av 'lønnskonto'- klassen og 'Boligspar' - klassen. Skissemessig vil det i C++ se ut som i Programeksempel 2 (det spesielle ordet 'public' kommer av krav fra hvordan vi må skrive C++, og er ikke viktig her).

Objekter av klassen lønnskonto og Boligspar konto er hver summen av 'konto'-egenskapene (data og prosedyrer i konto-klassen) pluss sine egne tilleggsdeklarasjoner. Objekter av både Boligsparklassen og lønnskonto-klassen vil begge ha data som beskriver navn og rentefot, men bare Boligsparobjektene har variabelen 'årlig_sparebeløp', osv.

Slike subclasser kan godt defineres over flere nivåer, slik at hvis vi hadde to typer av lønnskonti (offentlig-ansatt og privat-ansatt), kunne vi igjen ha de felles egenskapene i en klasse 'lønnskonto' og så lage subclasser av denne slik at vi får følgende hierarki i Programeksempel 3:



Klassediagram

Objekter av offentlig-ansatt har da "arvet" alle egenskapene til konto og lønnskonto i tillegg til de egenskapene som er deklartert i 'offentlig-ansatt' - klassen.

I noen språk, som Simula og Java, kan én klasse bare være subclasse til én annen klasse, dvs. bare *direkte* arve egenskapene til en annen klasse (selv om det i eksempelet ovenfor indirekte også kan arve fra den klassen klassen er subclasse til - slik type2 arver fra lønnskonto som igjen har arvet fra konto).

I siste versjon av C++ er denne restriksjonen opphevet slik at en klasse kan være subclasse direkte av flere andre klasser. Et eksempel på nytten av dette kan være at det systemet man bruker tilbyr ferdig kompilerte klasser, f.eks en klasse som inneholder mekanismer for effektiv sortering og en annen klasse som inneholder visse statistikkrutiner. Hvis vi så skal modellere f.eks pasienter i et større sykehus hvor vi både har behov for effektiv sortering av disse samt å foreta visse statistiske undersøkelser, kan vi gjøre vår pasientklasse subclasse til begge disse to ferdigkompileerte klassene.

Det kan være fornuftig å tenke på en klasse som en verktøyboks med et spesielt sett med verktøy. Når vi så lager en ny klasse, ser vi rundt oss og ser om noen allerede har laget de verktøy vi trenger, og i så fall gjør vi den klassen vi nå skriver til subclasse av de 'verktøy-boksene' vi trenger.

En avsluttende bemerkning om bruk av subclasser. Det er ofte, men ikke alltid man har behov for denne mekanismen. Man skal være forsiktig å definere for mange lag med subclasser. Hvis man definerer alt for mange små klasser og lar den ene være subclasse til den neste i en lang kjede, mister man ofte oversikten.

4.2 Redefinering av operasjoner - virtuelle prosedyrer

I forrige avsnitt ble det forklart hvordan en subklasse som Boligspår-konto arvet alle egenskaper fra klassen den er subklasse til, både datadeklarasjoner og prosedyrer. I visse tilfeller kan det være gunstig å redefinere en eller flere av de prosedyrene man arver.

Det første tilfellet kan illustreres med bankeksemplet vårt. Anta at alle ulike bankkonto-typer med unntak av Boligspår-konti beregner renter på samme måte. Det ville da være dumt om vi måtte skrive denne samme renteberegnings-prosedyren i alle de ulike kontotype-klassene bare fordi en eneste av subclassene krevet en egen løsning. Langt bedre ville det være om den (nesten) felles måten å beregne renter på bare ble skrevet en eneste gang i den felles klassen 'konto', og at bare i den subclassen som hadde behov for det, redefinerte denne (erstattet den) med sin egen prosedyre med samme navn.

Dette kan gjøres med såkalte 'virtuelle' prosedyrer. Hvis en prosedyre får det reserverte ordet 'virtual' før navnet når den deklarerer i den øverste klassen (i vårt eksempel i klassen konto), blir den redefinierbar i eventuelle subclasser. Når vi så i Boligspår-klassen skriver den annerledes renteberegningsprosedyren med samme navn for Boligspår-konti, blir det den som gjelder i alle de Boligspår-objektene vi oppretter. Det som i begynnelsen er litt overraskende, men som er helt nødvendig, er at det er ikke bare i de tilleggene som skrives i Boligspår-klassedeklarasjonen at denne nye prosedyren nyttes, men også i de andre prosedyrene som kaller på denne virtuelle prosedyren.

```
class konto
{ .....
  int saldo, .....;
  virtual int beregn_renter ()
  { .. 'vanlig renteberegning' .. };

  void årsoppgjør()
  { ..
  .....
    saldo = saldo + beregn_renter();
  .....
  };
};

class Boligspår: public konto
{ .....
  int beregn_renter ()
  { .. 'beregnet på Boligspår-måten' .. };
  .....
};
.....
```

Programeksempel 3.

I eksemplet vil et Boligspår-objekt nytte Boligspår-varianten av 'beregnet_renter' også i prosedyren 'årsoppgjør' som står deklartert i 'konto'-klassen. Når vi tenker nøyere etter, er dette logisk - skal vi først endre en prosedyre i et objekt, må vi gjøre det i hele objektet, ikke bare i noen deler. I vårt

eksempel vil selvsagt ikke renteberegningsrutinen bli endret i noen av de andre klassene som f.eks i lønnskonto-klassen, og heller da selvsagt ikke i prosedyren 'årsoppgjør', fordi 'beregner' ikke er redefinert i disse klassene. Det er alltid slik i C++ at når vi oppretter et objekt av en klasse, vil den forbli et objekt av denne klassen hele sin levetid.

Selv om det overnevnte eksemplet på bruk av virtuelle prosedyrer er viktig, finnes det en mer vesentlig bruk av denne mekanismen. I de fleste systemklasser som skrives defineres svært mange prosedyrer som virtuelle. Disse prosedyrene er som oftest meget små og gjør bare det helt minimale eller ingen ting i det hele tatt. Som i bankeeksemplet er ideen da at disse virtuelle prosedyrene kalles av andre prosedyrer som definerer mere sammensatte handlinger i systemklassen.

Særlig når de virtuelle prosedyrene i systemklassen er tomme, har vi en interessant tilfelle. Disse virtuelle prosedyrene står da for potensielle handlinger som brukeren selv kan fylle ut og på den måten modifisere de mer sammensatte handlingene. Hvis vi ønsker å redefinere disse virtuelle prosedyrene, tar vi, når vi har laget en subklasse av systemklassen, bare å skriver vår versjon av disse handlingene.

Et eksempel kan klargjøre dette. Som tidligere nevnt, vil en objektorientert implementasjon av et grafisk vindusystem bestå av en rekke systemklasser, en for hver av de ulike feltene vi kan ha på skjermen. Et slik felt kan være enkle tekstbokser som nyttes til å skrive inn énlinjers tekster eller tall. En slik systemklasse 'tekstboks' kan ha definert en virtuell prosedyre 'mouse_click' som av en annen rutine i vindusystemet kalles hver gang brukeren klikker med musa innenfor rammene til tekstboksen. I systemklassen er denne virtuelle prosedyren tom, og gjør følgelig ingen ting hvis den blir kalt.

Hvis derimot en applikasjonsprogrammerer ønsker beskjed når en bruker klikker i denne tekstboksen, skriver han selv bare en egen prosedyre 'mouse_click' i en subklasse av systemklassen for tekstboks. Nå er det denne nye prosedyren som blir kalt hver gang brukeren klikker med musa i tekstboksen, og programmereren kan da gjøre de ønskete handlinger (tolke teksten eller tallet i boksen, skrive ut en melding til brukeren på skjermen,...)

Disse virtuelle prosedyrene i systemklasser blir altså muligheter som programmereren kan nytte seg av eller ignorere. De er de punktene hvor systemklassen kommuniserer med applikasjonsprogrammereren og tillater at hun eller han kommer inn i systemets handlinger og evt. modifiserer systemets oppførsel etter eget behov.

5 Objektorientert analyse og design, en konkret metodikk

Vi har tidligere drøftet viktige enkeltspørsmål ved objektorientert programmering og systemering. Det er imidlertid for springende som til å være en "bruksanvisning". I dette kapittelet skal det derfor gis en punktvis metode for analyse og design. Det må imidlertid advares mot alltid å følge denne slavisk og rett fram fra pkt. 1 til 8. Som oftest vil man oppdage at man gå tilbake i denne rekken og gjenta noen av punktene etter hvert som forståelsen av problemområdet øker. Dels vil man også oppdage at man ofte kan slå sammen noen av punktene i en konkret analysesituasjon. Med disse advarsler, presenteres flg. åtte punkter .

1. Finn typisk brukere og bruk, samt kommunikasjon med andre systemer
2. Identifiser klassene
3. Bestem ansvarsområdet for de enkelte objektene
4. Bestem samarbeidspartnere for de enkelte objektene

5. Bestem arv (subklassene)
6. Lag samlinger av klassene (delsystemer)
7. Sett opp protokoller for kall fra hver klasse til dens samarbeidspartnere.
8. Lag funksjonelle krav til operasjonene (prosedyrene) i hver klasse

1. Bestem først ulike brukergrupper av systemet. Disse kalles gjerne aktører. Tegn typiske bruks-situasjoner med informasjonsflyt mellom tenkt system og hver aktør. Se også på mulige konflikter mellom brukerkravene. Oftest finner vi at ulike brukergrupper kan ha høyst ulike krav til et system og at deres ulike krav springer ut fra deres plassering i bedriften, og ikke nødvendigvis er basert på en misforståelse. Når slike krav fra ulike brukergrupper er reelt i strid med hverandre, bør man bli klar over dette - og ta det opp i prosjektet. Det er bedriften og ikke systemet som skal avveie ulike gruppers arbeidssituasjon legitime interesser mot hverandre (jfr. arbeidsmiljø loven, drøftinger med fagforeninger o.l.). Analysens jobb er da evt. å finne slike krav som kan være i strid med hverandre - ikke avgjøre slike konflikter. Overser man dette punktet, kan man risikere at en eller flere brukergrupper er så negative at systemet faktisk ikke blir tatt i bruk - og hele systemutviklingen har vært forgjeves.

Ikke bare aktørenes bruk av systemet må kartlegges, også systemets grensesnitt med andre systemer må beskrives. Av og til er dette enkelt, som at det nye systemet enten leser fra eller skriver filer til et annet system. Analysen er da enkel. Hvis derimot de to systemene står i direkte kontakt med hverandre og utveksler meldinger med data eller prosedyre-kall til hverandre, kan det ofte være lurt å modellere det andre systemet som en (menneskelig) aktør - og bruke en analyse som om det andre systemet var som enhver annen brukergruppe.

Finn så de mest kritiske krav til systemet og implementer disse først. Grader kravene etter viktighet (må ha, viktig, greit å ha). Ta bare med i versjon 1 de krav man 'må ha'. Få fastspikret hvilke krav som skal inn i versjon nr. 1. Erfaringsmessig en flytende krav til hvilken funksjonalitet som skal inn i versjon nr. 1 av systemet en av de viktigste grunnene til at systemutviklingsprosesser mislykkes (jfr. Tress90prosjektet i Trygdeetaten). Få også ledelsen interessert i prosjektet og med på beslutningene.

2. Identifisering av klassene gjøres kanskje best ved å ta utgangspunkt i mest brukte ting eller begreper, dvs. substantivene, innen problemområdet. Noen av disse vil være opplagt kandidater til å implementeres som klasser. Lag derfor først en vanlig skriftlig beskrivelse av problemet og de krav man har til systemet (en kravspesifikasjon) . Understrek deretter substantivene i denne teksten, og klassifiserer dem i tre grupper: Helt sentrale, som forekommer oftest i teksten, noe mer underordnede og til sist de tvilsomme som forekommer meget skjelden . Husk å ikke bare ta med substantiver fra selve problemområdet, men også mulige objekter fra selve løsningen, som utskriftsenheten, dialogvinduer på skjermen osv.

Kategorier av klasser, som bankkonti generelt, kan også være mulige kandidater - de kan bli klasser som man senere kan lage subklasser fra. Start med utgangspunkt i de klare kandidatene som forslag til klasser, og utvid evt. tilfanget etter hvert som du ser ønskeligheten av flere klasser.

3. Ansvarsområdet for klassene er hvilke enkeltdata og hvilke aksessprosedyrer de skal inneholde. Mens datadelen ofte kan være relativt lett å bestemme, er funksjonaliteten ikke like opplagt. Start også her med å skrive opp de mest opplagte funksjoner en klasse skal inneholde. Når man har gjort dette for samtlige klasser, vil man kanskje se at noen funksjoner mangler, og disse må så 'gis' til den klassen som mest naturlig kan ta en slik funksjonalitet. Som et alternativ må vi nå revidere og evt. utvide antall klasser.

4. I denne fasen, hvor man ser på samarbeidet (kommunikasjonen) mellom de enkelte klassene, vil man studere hvilke klasser som kaller aksessprosedyrer i andre klasser. Tegn klassene og kommunikasjons-piler for hvert kall. Man vil da kunne oppdage at noen av prosedyrene bør overføres fra en klasse til en annen klasse for å minimere kall mellom mellom klassene (slik vi tidligere har drøftet).

5. Bestemmelse av subklasser og arv går dels på det vi gjorde under pkt.1 da vi bestemte grupper av objekter. I denne fasen prøver vi å finne klasser som hittil i analysen har store fellesdeler i sin funksjonalitet. Disse fellesdelene mellom to eller flere klasser trekkes så ut som en felles, abstrakt klasse, som man så danner konkrete klasser ut fra ved sub-klasse mekanismen. Det er vanlig at i det kjørende programmet ikke dannes objekter av slike felles, abstrakte klasser - deres 'eneste' funksjon er å være basis for subklasser, som det vil bli dannet objekter fra.

6. Det å lage delsystemer er ikke det samme som å lage subklasser og arvehierarkier slik vi gjorde i forrige punkt. Hensikten nå er å lokalisere grupper av klasser som har stor grad av kommunikasjon seg imellom, som som har relativt liten kommunikasjon med de øvrige klassene. En slik gruppe av klasser kan vi betrakte som et eget subsystem og på mange måter betrakte som en enhet. Oppdeling av hele systemet i subsystemer er spesielt viktig ved større systemer. F.eks kan vi gi hvert slikt subsystem et eget navn, det kan også være hensiktsmessig å deklare en egen klasse som sørger for all kommunikasjon mellom subsystemet og omverden, og vi kan også legge disse klassene på en egen fil for separat kompilering. En slik enhet er ofte en velegnet enhet som ansvarsområde for en programmerer.

7. Oppsett av protokoller mellom de enkelte klassene vil si at vi på en mest mulig entydig måte (ofte ved hjelp av formell, logisk notasjon) setter krav til hvilke relasjoner vi har mellom parameterne når en aksessprosedyre kalles, og hvilke relasjoner som skal gjelde mellom øvrige data/utparameterne etter at prosedyren er utført. Det er altså ikke alltid nok å sette krav til hver enkelt parameter separat, men også til sammenhengen mellom disse og eventuelle globale datastrukturer som blir endret av prosedyren.

8. Det å lage funksjonelle krav til den enkelte prosedyre, er en velkjent aktivitet. Mens vi i forrige punkt satte krav til selve grensesnittet til prosedyren, settes her funksjonelle krav til den enkelte prosedyre. Dels kan dette gå på metodevalg for implementering av en bestemt beregning, men dels kan det gå på detaljert spesifisering av den interaksjon som prosedyren f.eks har med en (interaktiv) bruker.

-- o O o --

Denne skrittvisе metoden går selvsagt inn i en større sammenheng og startes ikke opp før vi har gjort en rimelig forstudie i samarbeid med relevante brukergrupper. Det antas da at vi i det minste har en grov systemspesifisering. I løpet av denne prosessen må man selvsagt også kommunisere med brukerne, slik at man har mulighet til å få justert designen underveis.

Det er også verdt å legge merke til at vi med overskriften har slått sammen analyse- og designfasen. Ved en objekt-orientert tilnærming er dette mer naturlig enn ved andre teknikker fordi den notasjonen man nytter helt fra analysefasen, via design-fasen og til programmeringen er den samme: Objekter og samarbeid mellom disse. Ved andre analyse- og design-metoder går man gjerne over fra f.eks en type notasjon med bokser og piler til en relativt annerledes notasjon- og begrepsverden når den detaljerte designen spesifiseres. En stor grad av modellnærhet med den virkelige verden fra de første analysene til den ferdige koden er en av de store fordelene ved objektorientert systemutvikling.

Det overstående beskriver en 'manuell' analyse med papir og blyant. Det har de siste årene blitt utviklet både en notasjon for de diagrammer man lager under en slik analyse, UML (Unified Mo-

delling Language). UML er meget godt forklart i Fowler & Kendal: "UML Distilled". Det er også dataverktøy som støtter denne noasjonen, f.eks. RationalRose (<http://www.numerica-tascon.no/> og <http://www.rational.com>). I RationalRose kan man bl.a. fra de diagrammene man lager av klassene generere kode-skjeletter i valgt programmeringsspråk (C++, Java, Visual Basic) - og man kan også analysere gammel kode i slike språk og generere klassesdiagrammer fra disse.

6 Objektorienterte brukerdialoger og objektbaserte komponenter

En ting er at selve programmeringen er objektorientert, noe helt annet er at det programmet under eksekvering har en måte å kommunisere som kan kalles objektorientert. Hva en objektorientert brukerdialog er kan vi lettest forstå ved først å se på det motsatte, en kommandodrevet brukerdialog.

Et godt eksempel på en kommandodrevet brukerdialog er kommandolinjen i DOS. Til DOS gir man kommandoer som: 'copy', 'dir', 'mkdir', osv. Man gir først et verb, en kommando, og som parametre til denne gir man så evt. de data, de gjenstander som kommandoen skal virke på. I en kommandodrevet brukerdialog gir man først verbet og så objektet/objektene det skal virke på. De er flere ulemper med dette for brukeren. For det første er det ofte nødvendig for brukeren at han eller hun husker den relevante kommandoen. Dernest kan man lett få vanskelige feilsituasjoner ved at man har oppgitt feil objekter (f.eks filer) eller gitt dem i gal rekkefølge. Det kan også være vanskelig å avbryte en slik kommando når man først har startet på den. Fordelen med et kommandodrevet brukergrensesnitt er at det er meget raskt for en erfaren bruker av vedkommende system.

I en objektorientert brukerdialog, velger man alltid gjenstanden(e) man skal operere på først - vi velger objektet(-ene). Dernest bestemmer vi handlingen. Fordelen med denne rekkefølgen er at hvis vi først har valgt objektet, kan systemet finne ut hvilke operasjoner som er mulige på dette objektet, og f.eks presentere disse mulige operasjonene i form av en meny (hvis brukeren ber om dette). En fordel med denne angrepsmåten er at den er langt lettere å lære for en uøvet bruker samt at den gjør det også meget lettere for en øvet bruker å lære et nytt system.

Noen eksempler på objektorientert brukergrensesnitt er f.eks MS Word under Windows hvor man alltid først må velg et utsnitt av teksten før man kan velge operasjoner som fjern, klipp ut, omformatter, osv. Operasjonen får da effekt for det valgte tekstområdet. Et annet eksempel kan være Windows 95, hvor man for å få fjernet eller kopiert en eller flere filer, først må velge disse, og så gi ordren.

Av det overstående kan man lett få det inntrykk at et objektorientert grensesnitt er det samme som et vindusbasert grensesnitt. Det er en feiltakelse. Det er lett å lage et kommandobasert brukergrensesnitt med vinduer. Poenget er heller at det er meget vanlig å lage et objektorientert brukergrensesnitt med vinduer etc. og at objektorienterte brukergrensesnitt i all hovedsak har blitt laget ved hjelp av slike grafiske vindussystemer.

Det er imidlertid ikke lett, og ikke nødvendigvis lurt, alltid å være 100% objektorientert i brukergrensesnittet. Selv i MS Word starter man med en kommando: 'åpne fil', hvor man først velger handling og deretter den filen man skal tekstbehandle. Ikke desto mindre er det meget sannsynlig at en slik objektorientert brukermodell, forøvrig den som spesifiseres som IBM's SAA brukermodell, har klare fordeler i brukervennlighet for alle med unntak av meget øvede brukere av vedkommende programprodukt.

En ny trend er at man kan kjøpe ferdige komponenter (VBX, OCX, ActiveX, JavaBeans) som man inkludere i sitt program. Typisk er dette spesialiserte kontrollere (mer avanserte enn trykk-knapper og list-bokser) som man nytter i et dialogvindu med sluttbrukeren. En slik kjøpe-kontroller kan f.eks være en tabell-håndterer, en som framviser spesielle data som f.eks kartinformasjon e.l.

for sluttbrukeren, foretar et databasesøk eller en som håndterer tekster og editering av tekst bedre enn standard-produktene fra kompilator-leverandøren. Dette er objekter med data og prosedyrer som er så generelle at de kan nyttes i mange sammenhenger.

7 Objektorienterte databaser

Mange snakker om objektorienterte databaser, men det er ikke enkelt å komme opp med en entydig definisjon. Selve begrepet er også problematisk. Databaser tar jo sitt utgangspunkt i et skille mellom program og data, og at det er datamodellen av problemet som lagres i databasen - den skulle følgelig være fri for program. Objektorientering på den andre siden tar sitt utgangspunkt i en helhetlig beskrivelse av de 'gjenstandene' vi modellerer i objekter som både har data og tilhørende operasjoner i en enhet - de to delene er uadskillelige.

En annen grunn til at objektorienterte databaser enda ikke har en klar definisjon, er at dette er et felt i rivende utvikling. Om noen få år er det å forvente at begrepet vil være bedre avklart. Det man kan si i 1997 at det synes klart å være to tilnæringsveier til det å lage objektorienterte databaser; den første kommer fra relasjonsdatabase-verden og den andre fra programmering-språkene som C++.

Det som fra relasjonsdatabase-verden og SQL synes å fremstå som objektorienterte databaser, er databasesystemer, som i tillegg til datamodelleringsverktøy og et sett med ferdige, ofte lavnivå systemoperasjoner (verb) som virker på denne datamodellen (select, update, delete ...), også tilbyr generelle programmeringsspråk med vanlige konstruksjoner som if-tester, løkker, variable og parametre til prosedyrer. Med disse språkene kan programmereren selv lage aksessrutiner (prosedyrer) til databasen. Slike prosedyrer blir lagret sammen med databasen på tilsvarende måte som datamodellen.

Det som blir fordelen med slike brukerdefinerte operasjoner i tillegg til de (lavnivå) operasjonene som alle databasesystemer tilbyr, er at uansett hvor 'gode' slike systemrutiner er, så er de generelle og ikke tilpasset den konkrete problemstillingen vi har. Når en applikasjonsprogrammerer kan definere sine egne operasjoner som en sammensetting av disse systemoperasjonene, får vi en enhet av data og operasjoner som vi kan kalle objekter.

Disse prosedyrene kan vi da si modellerer høyere nivå operasjoner på logiske deler av databasen. Vi kan da betrakte databasen som bestående av objekter av en eller flere klasser hvor datamodellen av en slik klasse sammen med de operasjonene som tilhører disse data. Adgang til data i databasen bør/skal da bare skje via disse aksessprosedyrene. SQL-server/Sybase og Ingres et to eksempler på produkter som har mulighet for slik programmering av aksessprosedyrer til databasen.

Det er imidlertid klart et det er et omstridt spørsmål hvor mye av logikken som bør ligge ute i slike aksessprosedyrer og hvor mye av logikken som skal ligge i det vanlige programmet som aksesserer databasen. Det arbeides også intensivt med disse problemene og andre spørsmål, og det er å forvente at begrepet 'objektorienterte databaser' fra relasjonsdatabase-verden og SQL, vil få et endret innhold om få år - bla. er det kommet en helt ny og utvidet standard 'SQL 3'. I denne kan man bl.a ha elementer i tabellene som ikke bare er enkle datatyper, som i dagens SQL, men sammensatte objekter. Man kan også lage subtabeller av en tabell. Med tid og stunder vil nok deler av forslagene i SQL3 bli implementert i de ulike leverandørenes produkter.

Fra objektorienterte språk som C++ kommer ideen om å kunne lagre ikke bare datadelen av et objekt, men også pekere objektene imellom og prosedyrene i objektene. Objekter som vi på denne måten lagrer i en database for at de skal overleve fra en programeksekvering til den neste, kalles persistente objekter. Det synes klart at det ikke er alle objekter som vi ønsker eller trenger skal være persistente, og noen objektorienterte databasesystemer har derfor innført eksplisitte kommandoer

for å navngi hvilke objekter som skal være persistente, samt operasjoner for å lagre (skrive) og fremhente (lese) slike objekter i en database. Andre systemer sørger for automatisk lagring av alle objekter av klasser som er subclasser av en bestemt basis, persistent klasse - alternativt at vi bestemmer at et bestemt objekt skal være persistent eller ikke når det opprettes med **new** operatoren.

Et annet viktig poeng i tillegg til at vi ønsker å lagre objektene med deres data og prosedyrer, er at vi også må lagre den strukturen og relasjonene som foreligger mellom objektene - dvs. vi må også kunne lagre pekervariable. Når vi vet at en peker internt under eksekveringen representeres med den maskinadressene i hurtiglageret hvor det objektet som pekes på ligger, ser vi at disse maskinadressene ikke kan brukes når vi skal lage objekter i en database. Når eksempelvis to objekter som peker på hverandre lagres i en database og senere leses tilbake til programmet, er det ikke sannsynlig at de plasseres på samme maskinadresse som forrige gang de lå i hurtiglageret. De gamle pekerne har derfor gal verdi - peker det 'gamle' stedet. Vanlig er det derfor at alle objekter i systemet får et unikt nummer, og at det er via disse nummerene (og en tabell som for hver eksekvering knytter sammen disse nummerene og aktuelle maskinadresser) at vi kan lagre objektene og gjenskape pekere til disse.

Distribuerte databaser, ulike versjoner av prosedyrene i objektene og backup/recovery er også vanskelige spørsmål når det gjelder objektorienterte databaser med den tilnæringsmåten at man utvider et eksisterende programmeringsspråk. Det finnes idag flere slike produkter, som Gemstone, ObjectStore, O2 og ONTOS på markedet.

Objektorienterte databaser er gjennomgående raskere og kan modellere langt mer komplekse datastrukturer enn relasjonsdatabaser, men mangler den fleksibilitet og standardisering som relasjonsdatabasene har fått via SQL. For å få et tilsvarende spørrespråk som SQL, har de ofte definert et SQL-lignende spørrespråk hvor man ber om å få den mengden av objekter fra databasen som tilfredstiller en bestemt betingelse. Pekere kan da brukes inne i f.eks en SELECTsetning. Anvendelsesområdene for disse databasesystemene er særlig kompliserte datastrukturer og/eller der korte responstider er påkrevet, som man f.eks finner ved teknisk konstruksjon (CAD).

8 Avsluttende bemerkninger

I det foregående har objektorientert teknikker vært beskrevet som en enhet mellom data og tilhørende operasjoner. Det viktige er at en slik klasse danner et begrep som vi kan bruke videre når vi systemerer og programmerer vårt system. Når vi først har definert/bestemt oss for å ha en klasse som 'konto' i vårt system, kan vi i alt videre arbeid med systemet tenke og snakke om kontoer og handlinger på disse. Vi har hevet tankegangen og resonneringen rundt løsningen fra programmeringsspråkets ord og variabeltyper og opp til begreper som hører hjemme i den problemverden hvor vi skal lage en programløsning.

Det er denne tankegangen og begrepsdannelsen som er viktig. Objektorientering er ikke knyttet opp til spesielle språk som Simula, Java, Smalltalk eller C++ . Med meget stor påpasselighet skulle det være mulig å skrive 'objektorientert' i f.eks Fortran. Det må imidlertid bemerkes at det er en uvurderlig hjelp å ha et programmeringsspråk som hjelper oss istedenfor delvis å bli motarbeidet av språkets begrensninger. Det er derfor ingen grunn til ikke f.eks å ta steget fra C til C++, fra Pascal til Object Pascal,..osv. Men for å understreke hovedpoenget en siste gang : Det er ikke skifte av programmeringsspråk som er det viktigste , det er hvilke begreper og filosofi vi bruker når vi bygger opp strukturer i programmet, som er vesentlig.

Et siste avsnitt som en liten advarsel. Objektorientert systemering og programmering løser ikke alle problemer. Det er også slik at fordelene ved objektorientert programmering først viser seg

ved programmer som er over en viss minstestørrelse - si mer enn 1000 linjer kode. Har man et kort og enkelt problem kan det være å skyte spurv med kanoner å definere klasser for et så lite problemområde. Objektorientering er først og fremst et verktøy til å beherske middels store og de helt store problemene. Derfor vil også ikke fordelene ved objektorientert programmering lett komme fram i enkle eksempler på slike innføringskurs - så enkle problemer kunne nesten vært løst på en hvilken som helst måte.

9 Litteratur

- Martin Fowler, Kendal Scott “UML distilled”, Addison Wesley, sec. ed. 2000
- Craig Larman: “Applying UML and Patterns” Second edition, Prentice Hall, 2002
- Bruce Eckel: “Thinking in Java” Second edition, Prentice Hall, 2000
- Ivar Jacobson, Grady Booch, James Rumbaugh “The Unified Software Development Process”, Addison Wesley, 1999
- L. Matiassen, A. Munck-Madsen, P.A. Nielsen, J. Stage: “Objektorientert analyse og design” MARKO, Aalborg 1997, (ISBN 87-7751-123-9)
- Grady Booch “Object Oriented Design with applications”. Benjamin/Cummings 1991
- Rebecca Wirfs-Brock, Brian Wilkerson, Lauren Wiener “Designing Object-Oriented Software”, Prentice Hall 1990
- Bjarne Stroustrup: “The C++ Programming Language” Second edition, Addison-Westley, 1991