Klasser og subklasser i Simula

Skrevet av Stein Krogdahl
Redigert av Bjørn Kirkerud

29. mars 1989

Klassebegrepet i Simula har svært mange aspekter. Vi skal her se på muligheten til å danne såkalte “subklasser”, noe som i mange situasjoner kan være meget nyttig.

For å få tak i idéen med subklasser skal vi se på et eksempel. Anta at vi skal skrive et program for et forlag, der det skal lagres informasjon om forskjellige typer bøker. Det vi skal lagre om hver bok er **text** forfatter, **integer** sideantall og **real** pris. Ut over dette skal det lagres litt forskjellige ting, ut fra hvilken type bok det er. Det skilles mellom *skolebøker*, der det skal lagres **integer** klassetrinn og **text** skolefag, *fagbøker* der det skal lagres **text** fagområde, og *romaner* der det skal lagres **text** sjanger (kriminal, kjærlighet etc).

Vi tenker oss så at situasjonen er slik at det er ønskelig å bruke klasse-objekter til å lagre informasjonen om hver bok. En måte å løse dette på er å lage en klasse bok som i tillegg til fellesattributtene nevnt over, også har alle attributtene klassetrinn, skolefag, fagområde og sjanger, samt et attributt **text** boktype som kan ha verdiene `skolebok`, `fagbok` og `roman`. Når dette siste attributtet f.eks. har verdien `roman`, så vil bare attributtet sjanger være i bruk, mens attributtene klassetrinn, skolefag og fagområde vil stå ubrukte. Denne løsningen vil derfor ta unødig mye plass.

En måte å rasjonalisere denne løsningen på ville være å slå sammen attributtene skolefag, fagområde og sjanger til ett felles attributt som f.eks. kunne hete undertype. Attributtet klassetrinn ville imidlertid fremdeles ofte stå ubrukt.

Dette eksempelet er ganske enkelt, men om vi tenkte oss et reelt tilfelle ville det sikkert være en hel masse ting som skulle registreres spesielt for de forskjellige typene bøker, og da kan den løsningen som er foreslått over fort bli for plasskrevende, selv om vi gjør den type rasjonaliseringer som er antydet. I Simula er det derfor en egen mekanisme, *subklasser*, som gjør at denne type problemstillinger kan løses meget elegant. For å løse problemet med bøkene våre definerer vi da følgende klasser:

```
class bok;
  begin
    text forfatter;
    integer sideantall;
    real pris;
  end bok;

bok class skolebok;
  begin
    integer klassetrinn;
    text skolefag;
  end skolebok;
```

```
bok class fagbok;
  begin
    text fagomraade;
  end fagbok;

bok class roman;
  begin
    text sjanger;
  end roman;
```

Her sier vi at de tre siste klassene har klassen bok som *prefiks*[1], og de sies å være *subklasser* av klassen bok. Når f.eks. klassen skolebok har bok som prefiks betyr det at klassen skolebok er en videre utbygging av klassen bok. Klassen skolebok får altså for det første alle attributtene som klassen bok har, og i tillegg de som blir oppgitt eksplisitt i definisjonen av skolebok. Tilsvarende gjelder for de to andre klassene med bok som prefiks. Dermed får vi nettopp de klassene vi ønsket oss.

De tre subklassene av bok er å betrakte som helt vanlige klasser. Vi kan deklarere

```
ref(skolebok) rsb;
ref(fagbok)   rfb;
```

og vi kan si

```
rsb :- new skolebok;
rfb :- new fagbok;
```

Videre kan vi si både

```
rsb.sideantall := 250;
rfb.sideantall := rsb.sideantall + 100;
```

og

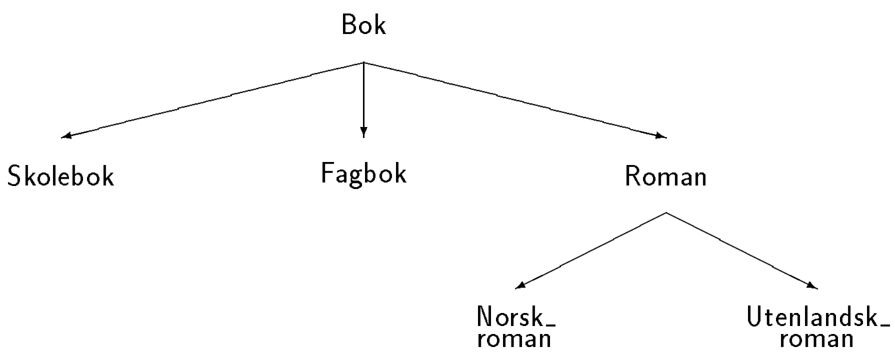
```
rfb.fagomraade :- "Huleforskning";
```

Fra før av er vi vant til at en **ref**(X)-variabel bare kan peke til objekter av klassen X. I forbindelse med subklasser blir det imidlertid løst litt på denne reglen, og det er dette som gjør subklasser til det fleksible verktøy det er. I tillegg til at en **ref**(X)-variabel kan peke til X-objekter, kan den også peke til objekter av alle subklasser av X (og subklasser av disse igjen). Om vi altså har en **ref**(bok) rb, så kan vi uten videre utføre rb :- **new** skolebok.

Vi kan også uttrykke denne nye reglen på en annen måte, nemlig ved å si at en skolebok jo også er en bok. Dermed kan selvfølgelig også en **ref**(bok)-variabel peke til en skolebok. Det som her er viktig rent språkteknisk er at alle objekter som en **ref**(bok)-variabel kan peke til har alle attributtene som finnes i klassen bok. Dermed kan vi helt trygt si f.eks. rb.sideantall, for enhver subklasse av bok har dette attributtet.

Før vi går videre skal vi knytte en kommentar til ordet *subklasse*. Denne betegnelsen kan synes noe merkelig, da objektene av en subklasse jo faktisk er større (har flere attributter) enn objekter av den originale klassen. For å forstå meningen med ordet *subklasse* må man legge an en annen synsvinkel, nemlig å lese ordet *klasse* som *mengde*. Man tenker seg da at klassen bok representerer mengden av alle bøker (altså mengden av alle objekter av klassen

[1]Ordet *prefiks* betyr noe som er satt foran (*pre* betyr *foran*).



Figur 1: *Et hierarki av klasser og sub-klasser.*

bok). Innen denne mengden har vi definert noen undermengder (sub-mengder), nemlig mengdene skolebøker, fagbøker og romaner. Fra denne synsvinklen er det rimelig å si at klassen av skolebøker er en subklasse av klassen av bøker, og Simula insisterer altså på at en skolebok både kan sees som et objekt av den snevre klassen skolebok og som et objekt av klassen bok.

At objektene av subklasser blir store er da rimelig ut fra at jo sterkere du har avgrenset den gruppen du snakker om, jo flere egenskaper er det interessant å trekke fram ved individene i gruppen. Det er f.eks. lite du kan si om en bok, den har bare en forfatter, en pris, og et antall sider. Spesifiserer du nærmere hvilken undergruppe av bøker du snakker om, er det straks flere ting å si om denne. Vet du f.eks. at det er en skolebok det er snakk om, kan du straks spørre om hva slags klassetrinn og fag den er for.

Klasse-hierarkier

La oss så gå videre med noen flere aspekter av subklasser. For det første kan vi også ha subklasser av subklasser. Vi kan f.eks. definere to subklasser norsk_roman og utenlandsk_roman av klassen roman. Et slikt system av klasser og subklasser kalles gjerne et *klasse-hierarki*, og dette tegnes gjerne som et tre (som vokser nedover, slik de gjerne gjør i databehandling!) der den ytterste klassen tegnes øverst, og der det er en linje fra hver subklasse og opp til dens prefiks-klasse. Om vi også tar med de to siste klassene vil vårt klasse-hierarki se ut som vist i figuren.

Ut fra denne tegningen er det lett å fastslå hva slags objekter forskjellige slags variable kan peke til: En **ref**(bok)-variabel kan peke til objekter av alle klassene, mens en **ref**(skolebok)- eller en **ref**(fagbok)-variabel bare kan peke til objekter av akkurat de respektive klassene. En **ref**(roman)-variabel kan derimot peke til objekter av klassene roman, norsk_roman og utenlandsk_roman.

Siden en variabel kan peke til objekter av flere klasser, er det i Simula også mulig å teste hva slags objekt variablen faktisk peker til. Anta at rb er en **ref**(bok)-variabel. Vi kan da si:

```
if rb is roman then ...
```

Denne testen vil gi **true** dersom rb peker til et objekt av akkurat klassen roman, men altså ikke om objektet er av f.eks. klassen norsk_roman. Dersom vi vil teste om rb peker til et objekt av klassen roman eller av en subklasse av roman, så må vi si:

if rb **in** roman **then** ...

Med det klasse-hierarkiet vi har her vil testene rb **is** skolebok og rb **in** skolebok være helt ekvivalente, siden skolebok ikke har noen subklasser.

Man skal være klar over at ordet *subklasse* ofte brukes i litt forskjellige betydninger. En måte å bruke dette ordet på er å si at subklassene til bok, det er skolebok, fagbok og roman. Det er imidlertid vanligere også å regne norsk_roman og utenlandsk_roman som subklasser til bok, og da si at de tre første er "umiddelbare" subklasser av bok.

Ofte regnes også en klasse som subklasse av seg selv. Da kan man f.eks. kort si at en **ref**(X)-variabel kan peke til objekter av alle subklasser av X. Hva som til enhver tid menes med ordet "subklasse" vil som regel greitt gå fram av sammenhengen. Man bruker ofte også ordet "superklasse" i betydningen at X er en superklasse av Y hvis og bare hvis Y er en subklasse av X.

Selv om et navn er brukt på et attributt i en (super)klasse, kan man godt bruke navnet om igjen på et attributt i en subklasse. Effekten blir da akkurat som ved bruk av samme navnet i blokker som ligger inne i hverandre, nemlig at deklarasjonen i subklassen er den som blir gjeldende så lenge man ser det hele inne fra subklassen.

Når det gjelder aksess til attributter er det slik at om vi f.eks. ser et roman-objekt gjennom en **ref**(bok)-variabel rb, så "ser objektet ut som" et bok-objekt. Vi kan altså si rb.sideantall, men rb.sjanger vil kompilatoren protestere på. Dersom vi vet at rb peker til et roman-objekt kan vi imidlertid være interessert i å få tak i attributtet sjanger, og en måte å gjøre dette på er slik (andre måter kommer lenger ned):

rb **qua** roman.sjanger

Nøkkelordet **qua** betyr altså "betraktet som". Når man på denne måten insisterer på at det som rb peker til skal betraktes som et roman-objekt, så vil systemet bak kulissene (under utførelsen) forsikre seg om at rb faktisk peker til et objekt av klassen roman (eller av en subklasse). Om dette ikke er tilfelle blir programmet avbrutt.

Så litt om tilordning. Her må vi passe på at vi ikke bryter regelen om at en **ref** (X)-variabel bare kan peke til objekter av klassen X og subklasser. Om vi har variablene **ref**(bok) rb og **ref**(roman) rr, så får vi lov til å si både rb :– rr og rr :– rb. Den første av disse er helt grei, fordi det rr peker på jo må være av klassen roman (eller subklasse), og det er derfor helt sikkert av en subklasse av bok.

Ser vi på rr :– rb blir det mere problematisk. Her kunne jo rb peke f.eks. til et skolebok-objekt, noe som ville føre til at rr kom til å peke til et slikt objekt. Dette ville stride mot reglen om at rr bare kan peke til objekter av klassen roman (og subklasser), og utførelsen av dette imperativet blir derfor ulovlig dersom rb ikke peker til et roman-objekt. Liksom for **qua** blir dette testet under utførelsen, bak kulissene, og programmet blir avbrutt om det ikke er tilfelle.

Denne type tilordning kan vi bruke til å hente attributter i en subklasse på en måte som ofte kan være mer rasjonell enn den over. Anta at vi igjen har en **ref** (bok)-variabel rb, og at vi etter å ha forvisset oss om at denne peker til et roman-objekt, gjerne vil hente fram diverse attributter deklarart i subklassen roman. Vi kan da deklarere en ekstra variabel **ref**(roman) rr, og gjøre det på følgende måte:

```
if rb in roman then
   begin rr :- rb; ... rr.sjanger ... rr.pris ... end;
```

Dermed slipper vi å si **qua** hele tiden, noe som vil spare både skriving og tid under utførelsen. I denne forbindelse bør også nevnes en spesiell form av **inspect**-imperativet i Simula som kan være velegnet nettopp i slike situasjoner. Den ser slik ut:

```
inspect rb
   when roman     do begin ... sjanger ... pris ... end
   when skolebok  do begin ... klassetrinn ... skolefag ... pris ... end
   otherwise         begin <Det som skal gjøres om det var ingen av delene> end;
```

Etter hver **when** C **do** kan man altså her uten videre snakke om alle attributtene i C. Man kommer også til "**otherwise**-grenen" om rb er **none**.

Klasser og subklasser med parametre

Så litt om parametere til klasser. Dersom det angis formelle parametere i en subklasse, så skal disse forstås som en utvidelse av den parameterlisten som allerede er definert for prefiks-klassen. Anta f.eks. at vi har følgende deklarasjoner:

```
  class A(x, y); integer x; real y;      begin ... end A;
A class B(t, u); text t; character u;  begin ... end B;
```

Klassen B har nå fire parametere, og når vi skal danne et nytt B-objekt kan vi f.eks. si:

```
rB :- new B(22, 3.14, "pling", 'w');
```

Klasser og subklasser må ha samme blokknivå

Det finnes en en regel i Simula om hvor klasser og subklasser kan deklareres. En klasse og alle dens subklasser må nemlig deklareres på samme "blokknivå". Det vil altså si at følgende programbit er ulovlig, da klassen B deklareres på et dypere blokknivå enn A:

```
begin
   class A; begin ... end A;
   ...
   begin
      A class B; begin ... end B;    ! Ulovlig!!;
      ...
   end;
end;
```

Virtuelle prosedyrer

Vi skal nå se på såkalte *virtuelle prosedyrer*. Et eksempel: Anta at det er en avgift på bøker, og at den skal beregnes litt forskjellig avhengig av om det er en skolebok, en fagbok eller en roman. En måte å legge dette inn i programmet på er å skrive en prosedyre **avgift** i hver av de tilsvarende subklassene, slik at disse beregner avgiften riktig for den tilsvarende type bøker.

Om vi har variablene **ref**(skolebok) rs, **ref**(fagbok) rf, og **ref**(roman) rr, og disse peker på tilsvarende objekter, så får vi beregnet riktig avgift for disse bøkene ved kallene rs.avgift, rf.avgift og rr.avgift. Om vi derimot har en **ref**(bok) rb som peker til et objekt av en eller annen av de tre klassene, er det ikke så kurant å få beregnet riktig avgift. rb.avgift er jo ikke en gang definert. Det måtte gjøres omtrent slik:

```
avgift:= if rb is skolebok then rb qua skolebok.avgift else
         if rb is fagbok   then rb qua fagbok.avgift   else
         ...
```

Det ville være greiere om det var definert en prosedyre i klassen bok som het avgift, slik at vi bare kunne si rb.avgift og dermed få beregnet riktig avgift. Dette er også mulig, og denne ville ta seg omtrent slik ut:

```
class bok;
  begin
    ...
    integer procedure avgift;
      begin
        if this bok is skolebok then avgift := ... else
        if this bok is fagbok   then avgift := ... else
        ...
      end;
    ...
  end bok;
```

Det dumme med denne metoden er at det blir masse kluss dersom avgiften er avhengig av attributter som ligger i subklassene. Simula har imidlertid en konstruksjon som kombinerer det gode i begge disse metodene, nemlig slik at avgiften enkelt kan fåes ved kallet rb.avgift, samtidig som prosedyrene kan skrives lokalt i hver subklassene. Dette gjøres som følger:

```
class bok;
    virtual: integer procedure avgift;
  begin
    ...
  end bok;

bok class skolebok;
  begin
    ...
    integer procedure avgift;
        avgift := ...;
    ...
  end skolebok;
```

Det blir helt tilsvarende for de to andre subklassene. I disse deklareres altså rett og slett prosedyrene, akkurat som i den første løsningen vi foreslo. Det spesielle ligger i klassen bok, der prosedyren avgift er spesifisert som *virtuell.* Når man har gjort en slik virtuell-spesifikasjon av en prosedyre, så må alle deklarasjoner ned gjennom subklassene med dette navnet være en prosedyre av samme type. Hver slik deklarasjon sies å være en "match" til den virtuelle deklarasjonen, også i det tilfelle at deklarasjonen gjøres i en subklasse av en klasse der det allerede er en slik match.

Når vi nå sier rb.avgift vil systemet si: Aha, avgift er en virtuell prosedyre, da skal jeg velge den "match" som svarer til den klassen objektet faktisk er av! Om variablen rb i øyeblikket peker på et roman-objekt, så vil prosedyren deklarert i subklassen roman bli valgt til utførelse, som altså nettopp var det vi ønsket.

Om rb hadde pekt til et objekt av klassen bok, så ville programmet stoppet, med en feilmelding: `"No match for virtual: avgift"`. Altså, den finner ingen match for avgift i et bok-objekt. Om det var ønskelig, kunne vi godt også deklarere en prosdyre avgift i klassen bok, slik at vi også fikk en match her. Denne ville da bli valgt i dette siste tilfellet, mens alle de andre tilfellene ville virke akkurat som før.

Anta nå at vi også har de to subklassene norsk_roman og utenlandsk_roman av klassen roman, og at det er deklarert en match til avgift i utenlandsk_roman, men ikke i norsk_roman. Vi har så f.eks. en **ref**(roman) rr (det vil virke likt om vi bruker rb), og sier rr.avgift. Om det er et roman-objekt rr peker på vil vi få utført den match som er deklarert i roman, og om det er et utenlandsk_roman-objekt, så vil vi få den som er deklarert her.

Men hva skjer om rr peker til et norsk_roman-objekt? Man kunne da kanskje tro et dette ga feil, men det gjør det ikke! Den match som brukes i dette tilfelle er den som er deklarert i roman. Når man skal avgjøre hvilken match som skal brukes, starter man altså å lete ved den klassen objektet faktisk er av, og man leter oppover gjennom superklassene til man finner en match. Om man ikke finner noe match er det feil, og programmet stoppes.

Klasser og subklasser med imperativer

Vi skal nå se på hva som skjer om vi har imperativer i klasser og subklasser.

Vi vet at når vi sier f.eks. new bok, så vil eventuelle imperativer i klassen bok bli utført umiddelbart, i forbindelse med genereringen av det nye bok-objektet. Om vi så har angitt imperativer i begge klassene bok og roman og sier new roman vil først imperativene i bok bli utført, og deretter de i roman. Imperativene i superklassene blir altså normalt utført først.

Det er imidlertid mulig å forrykke denne rekkefølgen noe, ved bruk av nøkkelordet inner. Vi deklarerer da klassene bok og roman slik:

```
class bok;
  begin
    <Deklarasjoner>;
    <Imperativer>;
    inner;
    <Imperativer>;
  end bok;

bok class roman;
  begin
    <Deklarasjoner>;
    <Imperativer>;
  end roman;
```

Om vi nå sier rr :– **new** roman vil først imperativene i bok foran **inner** bli utført, deretter vil imperativene i roman bli utført, og til slutt vil imperativene i bok etter **inner** bli utført. Nøkkelordet **inner** angir altså hvor subklassenes imperativer skal settes inn. Nøkkelordet **inner** kan brukes også i klasser med prefiks, men det kan bare forekomme en gang i imperativene i hver (sub)klassedeklarasjon.