2
Abstrakte datatyper.

Motiverende eksempel:
top-down udvikling af program 'mini-bank'

Strukturering af et program:
efter *data* eller *funktion* ?

Definition af en abstrakt datatype
og tilknyttede begreber.

Fænomener, begreber og begrebsdannelse.

Objekt-orienteret udvikling af
programmet 'mini-bank'.

© Kurt Nørmark, Aalborg Universitet, 1994.

Noter

Abstrakte datatyper udgør den allervigtigste ide i forhold til objekt-orienteret programmering. I denne forelæsning vil vi behandle emnet uafhængigt af programmeringssprog.

Topdown udvikling af programmet 'minibank' (1)

```
Program  mini-bank;

  Type Bankkonto-typedefinitioner af alle konto-arter;

  Transaktions-procedurer på de forskellige konto-typer;

begin
 while not fyraften
 do begin
       indlæs kontotype KT;
       udfør en transaktion på KT
    end
end
```


© Kurt Nørmark, Aalborg Universitet, 1994.

Noter

For at forstå egenskaberne af et objekt-orienteret program baseret på abstrakte datatyper er det nyttigt at sætte fokus på en anden programudviklingsteknik.

I denne forelæsning starter vi derfor med at studere et program udviklet ved såkaldt top-down programmering. Når et program udvikles top-down, startes med de overordnede dele af programmet, og dele deraf forfines trinvis, indtil alle detaljer er fuldt ud implementeret.

En modsætning til top-down programmering kaldes bottom-up programmering. I en bottom-up teknik starter man med at udvikle detaljer, og bygger op til en helhed. Man kan naturligvis forestille sig kombinerede top-down og bottom-up teknikker.

Topdown udvikling af programmet 'minibank' (2)

Eksempel på bankkonto typedefinition:

```
check-konto =
record
  id: KontoId;
  balance: Kroner;
  rente-sats: real;
  antal-udnyttede-checks: integer
end
```

udfør en transaktion på kontotype KT :

```
procedure transaktion(KT: kontotype);
begin
  case KT of
    check-konto:  check-konto-transaktion;
    aktionær-konto:  aktionær-konto-transaktion;
    pensions-konto:  pensions-konto-transakton;
    gevinst-konto:  gevinst-konto-transaktion;
     ....
  end;
end;
```


© Kurt Nørmark, Aalborg Universitet, 1994.

Noter

Her er vist et eksempel på en type-definition (af check-konti).

Endvidere er det centrale fragment af hovedprogrammet forfinet i forhold til forrige slide. Denne forfining har vi givet status af en procedure, *transaktion*, som naturligvis så skal kaldes af hovedprogrammet.

Procedurerne check-konto-transaktion, aktionær-konto-transaktion m.m., som kaldes i proceduren transaktion, vil vi på en naturlig måde kunne definere som lokale procedurer til transaktion.

Typen med navnet kontotype kan være en *enumeration type*, altså en type defineret ved

kontotype = (check-konto, aktionær-konto, pensions-konto, gevinst-konto, ...).

På næste slide viser vi en definition af proceduren check-konto-transaktion.

Topdown udvikling af programmet 'minibank' (3)

```
procedure checkkonto-transaktion;
var ck: check-konto;
transaktions-procedurer på check-konto
begin
  hent-konto(ck);
  indlæs transaktionstype TT;
  case TT of
    opret:  opret-check-konto(ck);
    hæve:  hæve-check-konto(ck);
    indsætte: indsætte-check-konto(ck);
    tilskriv-rente: tilskriv-rente-check-konto(ck);
    clear-check: clear-check(ck, check);
    nedlæg:  nedlæg-check-konto(ck)
  end
end
```


© Kurt Nørmark, Aalborg Universitet, 1994.

Noter

For hver type af bank-konti BKT antager vi, at der findes en procedure BKT-konto-transaktion, som i interaktion med brugeren udfører en af flere mulige transaktioner på en konto. Vi har vist proceduren for check-konti.

Man skal bemærke, at nogle af procedurerne tilsyneladende mangler noget information for at kunne virke. F.eks. er det ikke oplagt, hvilke beløb der hæves og indsættes på kontoen af hhv. hæve-check-konto og indsætte-check-konto. Vi antager, at disse procedurer spørger brugeren af programmet om den manglende information.

De enkelte transaktionsprocedurer antages at være lokale procedurer til den viste check-konto-transaktion, placeret under *transaktions-procedurer på check-konto.*

Bemærk, at vi "uden videre" ved at udføre den trinvise forfinelse har fået procedurer i tre indlejrede niveauer. Dette er meget karakteristisk for denne udviklingsteknik.

Tabel med oversigt over bankkonto-operationer.

Transaktions-type →

Konto-type ↓

	Hæve	Indsætte	Rente-tilskrivning	Oprette	...
Checkkonto	hæve-check-konto	indsætte-check-konto	tilskriv-rente-check-konto	opret-check-konto	
Aktionær konto	hæve-aktionær-konto	indsætte-aktionær-konto	tilskriv-rente-aktionær-konto	opret-aktionær-konto	
Pensions-konto	hæve-pensions-konto	indsætte-pensions-konto	tilskriv-rente-pensions-konto	opret-pension-konto	
Gevinst-konto	hæve-gevinst-konto	indsætte-gevinst-konto	tilskriv-rente-gevinst-konto	opret-gevinst-konto	
....					

© Kurt Nørmark, Aalborg Universitet, 1994.

Noter

Med den skitserede fremgangsmåde kræves der en transaktionsprocedure pr. konto-type pr. transaktion. Dette giver et stort antal forskellige procedurer, som vist i tabellen ovenfor. Nogle af disse procedurer har sikkert lighedspunkter, som det kunne være hensigtsmæssigt at samle på ét sted.

Hvis man ønsker at opretholde konto-type-definitionerne som hver sin record type, er det nødvendig at have en procedure for hvert felt i tabellen. Procedurerne er struktureret således, at hver *række* af procedurer er lokale til den bank-konto type specifikke transaktionsprocedure.

Man *kan* i Pascal arrangere tingene mere hensigtsmæssigt, nemlig ved at definere én type, som dækker alle konto-typer. En sådan record type skal have afdelinger med felter, der er specifikke for den enkelte konto-type. Sådanne records kaldes variant-records. Hvis ikke man allerede har gjort det i forbindelse med "grundbegreber", anbefales det at repetere dette begreb, som bl.a. findes i Pascal.

Opgave 1. Det blev observeret ovenfor, at vi primært har struktureret vores program efter *rækker* af operationer, og sekundært efter *søjler*. Demonstrer, at vi også kunne have valgt at gøre det omvendt; altså primært at strukturere efter arten af vores transaktion, og sekundært efter kontotype. Skitser det resulterende program. Hvilken af de to strategier (transaktions-art-først, kontotype-først) skal man foretrække, hvis vi antager, at der oftere skal tilføjes nye transaktioner end nye kontotyper? Og oftere nye kontotyper end nye transaktioner?

Opgave 2. Antag nu, at vi definerer en variant-record, hvor felterne *id* og *balance* er fælles for alle kontotyper, mens andre felter er specifikke for de forskellige typer af bankkonti. Vi vil nu forvente, at vi kan programmere nogle af bankkonto operationerne på et "fælles niveau". Skitser nu proceduren *transaktion* og dets lokale procedurer, og illustrer hvor de enkelte operationer nu hører hjemme.

Strukturering af et program: Efter *data* eller *funktion*

"Hvilke handlinger udfører programmet?" kontra
"Hvad udfører programmet handlinger på?"

Top-down udvikling af et program afspejler trinvis nedbrydning af et problem i mindre delproblemer og er som sådan med til at nedbringe "kompleksiteten" af problemløsningen.

En funktionel strukturering af et program er mindre stabil end en datastrukturering i programmets samlede levetid.

"Rigtige systemer" har ingen top.

Top-down udvikling og strukturering efter funktion virker hæmmende på udvikling af genbrugelige programkomponenter.

© Kurt Nørmark, Aalborg Universitet, 1994.

Noter

Mange af pointerne på denne slide er kondencerede udsagn fra Meyers: Object-oriented Program Construction, kapitel 4.

Definition af ‘abstrakt datatype’.

En abstrakt datatype er en mængde af værdier samt en mængde af operationer på disse værdier.

- Et program arbejder på **instanser** af en ADT, ikke på typen som så.
- Instanser af en ADT kaldes også for **objekter**.
- Udvalgte operationer på en ADT udgør det **interface**, som en **klient** af typen benytter til manipulation af instanser af typen.
- En abstrakt datatype skjuler, beskytter og indkapsler **data-repræsentationen** for klienter af typen.
- Ved programændringer i interne dele af den abstrakte datatype udgør operations-interfacet en "**brandmur**" mellem klient og data-repræsentation.

© Kurt Nørmark, Aalborg Universitet, 1994.

Noter

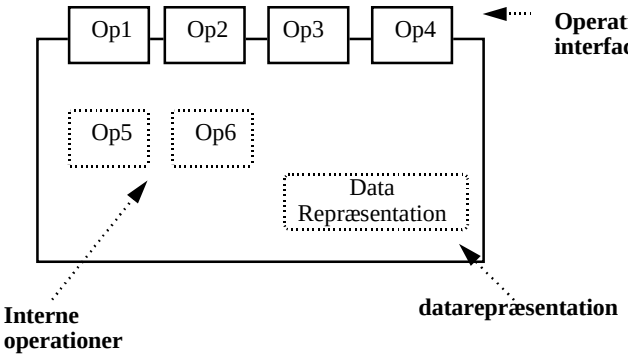
Operationsinterfacet opdeler operationerne på datatypen i interne og eksterne operationer. Mængden og naturen af interne operationer er af mindre betydning, sålænge de eksterne operationer fungerer, som det forventes (se nedenfor).

Tilsvarende er selve datarepræsentationen som regel skjult for klienter. Det betyder, at selv om en abstrakt datatype har en variable V som en del af sin repræsentation, kan man ikke udefra assigne til eller aflæse fra V direkte. Det skal ske gennem en operation.

Det dermed skabte operations interface udgør et *indirekte niveau* med hensyn til både aflæsning af og skrivning i instanser af typen. Dette indirekte niveau er meget værdifuldt. Hvis der sker ændringer i typens interne dele (i interne operationer eller i repræsentation) kan man nøjes med at ændre i eksterne operationers definition, og ikke i deres anvendelse rundt omkring i programmet. Det er på denne måde, at operationsinterfacet kommer til at spille rollen af en brandmur om datatypen.

Spørgsmålet melder sig, om det er muligt at angive, hvordan eksterne operationer i typen skal fungere, uden at programmere dem. Det kan man godt! På dette kursus vil vi studere to forskellige måder at *specificere* abstrakte datatyper. Allerede i næste forelæsning vil vi introducere den første af disse.

Bestanddele af en abstrakt datatype



PS1 -- Abstrakte datatyper

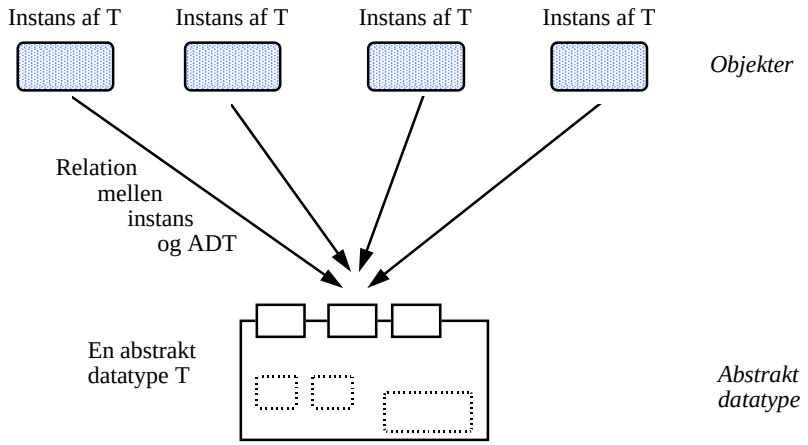
© Kurt Nørmark, Aalborg Universitet, 1994.

Noter

Man kan naturligvis opfinde mange forskellige tekstuelle såvel som grafiske notationer for abstrakte datatyper.

Her er bestanddelene af en abstrakt datatype illustreret grafisk.

En abstrakt datatype og dens instanser.



PS1 -- Abstrakte datatyper

© Kurt Nørmark, Aalborg Universitet, 1994.

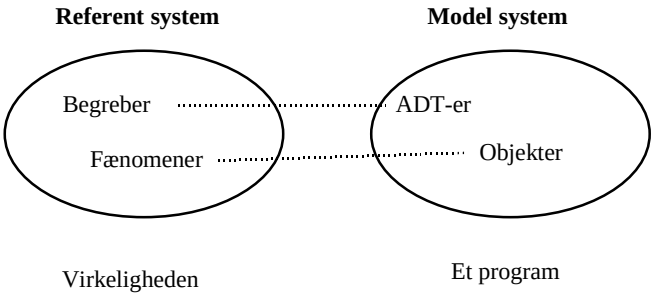
Noter

Ovenfor er abstrakte datatyper rektangulære, medens objekter har afrundede kanter. Forholdet mellem en abstrakt datatype og dets instanser er illustreret ved en pil fra instansen til dens ADT.

Instanser indeholder selvfølgelig objektets tilstand (i form af variable af konkrete, primitive typer, eller af andre abstrakte datatyper). Derimod repræsenteres objekternes operationer sædvanligvis ikke i objekterne, men i den tilhørende klasse. Dette kan lade sig gøre, idet operationer er ens for alle instanser af en given abstrakt datatype.

Hvordan finder man objekter?

Hvis et program modellerer en del af virkeligheden (såsom en bank), giver denne virkelighed ofte inspiration til hvilke objekter, og dermed abstrakte datatyper, et program skal indeholde.



© Kurt Nørmark, Aalborg Universitet, 1994.

Noter

Overskriften af denne slide skal læses som "hvordan finder man på, hvilke objekter der skal indgå i et program".

Pointen er, at hvis programmet, man skal skrive, behandler data, som er en del af vores virkelighed, giver begreber og fænomener (ting) i denne virkelighed inspiration til abstrakte datatyper og objekter.

Hvis programmet ikke retter sig mod virkeligheden, men måske internt mod maskinen, kan det være vanskeligere at finde på, hvilke abstrakte datatyper og hermed typer af objekter, der skal indgå i programmet.

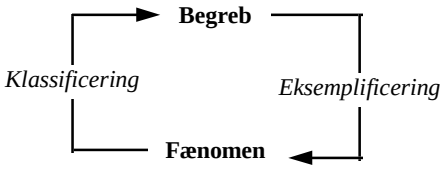
Fænomener og begreber

Fænomen:
En ting i den virkelige (eller "tænkte") verden med individuel eksistens.

Begreb:

Et begreb er karakteriseret af:

- *Navn*.
- *Intension* : mængden af egenskaber, der karakteriserer begrebet.
- *Extension* : mængden af fænomener, som er dækket (beskrevet) af begrebet.



PS1 -- Abstrakte datatyper

© Kurt Nørmark, Aalborg Universitet, 1994.

Noter

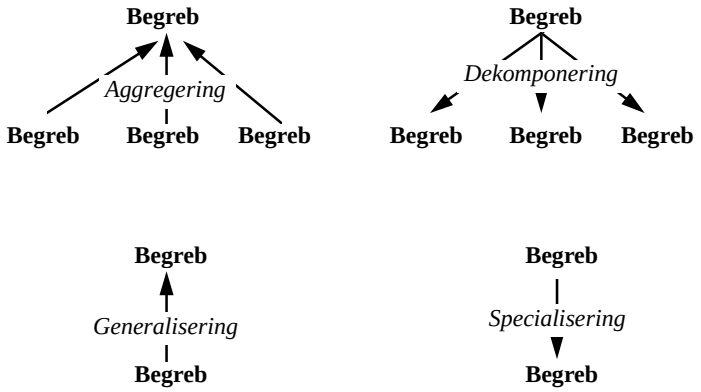
Begreber og fænomener hører til i den virkelige verden, i referent systemet. Abstrakte datatyper og objekter dannet ud fra abstrakte datatyper hører hjemme i programmeringsverdenen.

Emnet på denne slide er ikke beskrevet i bøgerne, som vi benytter på dette kursus. Hvis man ønsker at vide mere henvises der til f.eks. Jørgen Lindskov Knudsen og Kristine Stougaard Thomsen "A Conceptual Framework for Programming Languages" (DAIMI PB-192, April 1985).

Klassificering/eksemplificering er let at få øje på blandt begreber og fænomener i vores hverdag. Begrebet 'gravhund' kan have 'Fido' og 'King' som eksempler; omvendt klassificerer 'gravhund' de nævnte to eksempler på hunde.

Begrebsdannelse.

Hvordan danner man begreber ud fra andre begreber?



PS1 -- Abstrakte datatyper

© Kurt Nørmark, Aalborg Universitet, 1994.

Noter

De fire nævnte former for begrebsdannelse forekommer i par.

Aggregering samler et antal begreber i et nyt begreb. Dekomponering løsriver allerede samlede begreber i dets bestanddele.

Et generaliseret begreb betegner et bredere dækkende begreb end det specialiserede begreb. Ekstensionen af et specialiseret begreb er en delmængde af ekstensionen af det mere generelle begreb. Intensionen af det specialiserede begreb udvides typisk med nye egenskaber; det kan også forekomme, at eksisterende egenskaber redefineres (pålægges bånd) når man beskriver et specialiseret begreb.

Næste slide viser eksempler på de nævnte former for begrebsdannelse.

Eksempler på begrebsdannelse

Aggregering

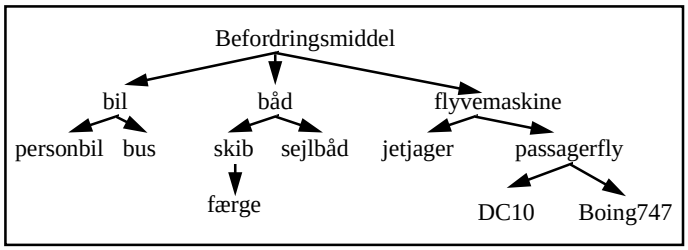
Begrebet en *flyvemaskine* består af

Et *cockpit*
En *krop*
Et *haleparti*
To *vinger*
Et antal *motorer*

En record **record**
f1: T1;
f2: T2;
f3: T3
end

er en aggregering af typerne T1 .. T3

Specialisering



PS1 -- Abstrakte datatyper

© Kurt Nørmark, Aalborg Universitet, 1994.

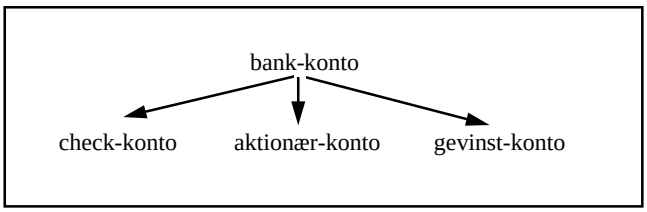
Noter

Indbyrdes generaliserede/specialiserede begreber danner et begrebshierarki.

Roden i hierarkiet er det mest generelle begreb. Hvis A->B i hierarkiet er B en specialisering af A (og A en generalisering af B).

Specialisering af abstrakte datatyper er et nøgleemne i objekt-orienteret programmering. Emnet vil komme til at spille en stor rolle senere i kurset. Den programmeringssproglige mekanisme, der understøtter specialisering af abstrakte datatyper, kaldes *nedarvning.* I senere kapitler af disse noter vil vi komme stærkt tilbage til dette.

Objekt-orienteret udvikling af 'minibank' (1)




```
ADT bank-konto
  id: KontoId
  rentesats: real
  balance: kroner
  opret (init-beløb: kroner)
  hæv (beløb: kroner)
  indsæt (beløb: kroner)
  tilskriv-rente ()
end bank-konto
```

```
ADT check-konto : bank-konto
  antal-udskrevne-checks: integer
  tilskriv-rente ()
  clear-check(beløb: kroner)
end check-konto
```

Tilsvarende definitioner af aktionær-konto og gevinst-konto

PS1 -- Abstrakte datatyper

© Kurt Nørmark, Aalborg Universitet, 1994.

Noter

Generaliserings/specialiserings hierarkiet af bankkonto-typer fastlægges som det første. Dette er et meget vigtigt trin i designet af et hvilket som helst objekt-orienteret program, der manipulerer bank-konti.

Hver af typerne, som indgår i hierarkiet, beskrives hvad angår repræsentation og operationer. Operationerne implementeres. På denne slide er implementationen af operationerne hverken vist eller antydet.

Der er benyttet en pseudo-syntax for abstrakte datatyper, som ikke er en del af noget programmeringssprog. Det eneste formål med denne slide er at illustrere principperne i objekt-orienteret udvikling, som et modstykke til den allerede viste top-down udvikling af mini-bank programmet.

En abstrakt datatype svarer til record-typen for bank-konto på den tidligere viste slide. Endvidere er udvælgelse af bank-konto operationen, som vist i check-konto-transaktion (se tidligere slide) implicit tilstede via mekanismer i det objekt-orienterede sprog.

Vi mangler "blot" at lave en (eller flere) hovedprogrammer med en passende bruger-interaktion på instanser af typerne. Et sådan er skitseret på næste slide.

Objekt-orienteret udvikling af 'minibank' (2)

```
procedure mini-bank-main-program
var K: bank-konto
begin
  while not fyraften do
  begin
    indlæs en bank-konto K
    indlæs en transaktions-type TT
    case TT of
      opret:  instantiering af et nyt bank-konto objekt
      hæve: indlæs beløb  B;   K.hæv(B);
      indsætte: indlæs beløb  B;  K.indsæt(B);
      tilskriv-rente: K.tilskriv-rente;
      clear-check: indlæs beløb  B; K.clear-check(B)
      nedlæg:  nedlæg bank-konto instans ;
    end
  end
end
```


© Kurt Nørmark, Aalborg Universitet, 1994.

Noter

Hovedprogrammet vist ovenfor er et af mange mulige hovedprogrammer, som kan programmeres med basis i de viste klasse-definitioner.

Flere steder i ovenstående program forekommer notationen K.*operation*(*parameter*). Denne notation kaldes "dot notation", og udtryksformen forekommer i mange objekt-orienterede programmeringssprog. Betydningen er at *operation* udføres på kontoen K med den angivne *parameter*. I mere konventionel Pascal-stil ville man skrive *operation*(K, *parameter*).

Det er værd at bemærke, at ovennævnte case-sætning kan bruges for alle typer af bank-konti *forudsat at alle typer af bankkonti har det samme sæt af operationer*. (Dette er ikke altid en realistisk antagelse. I så fald må man programmere en interaktion pr. kontotype).

I ovenstående program vil det være op til den objekt-orienterede omgivelse at udvælge kontotype-specifikke operationer. Når, f.eks. K.indsæt(B) kaldes i ovenstående program, udvælger systemet således den indsættelsesoperation, som er relevant på netop typen af kontoen K. Dette er en meget vigtig og karakteristisk egenskab ved objekt-orienterede programmeringsomgivelser. Senere i kurset vender vi tilbage til dette under betegnelsen "dynamisk binding".