Om objektmengder og Settools

Bjørn Kirkerud

Mars 1993

1 Objekter og objektmengder

Mange programmer holder styr på objekter og objektmengder. Når vi skal konstruere et program, kan det ofte være fornuftig ganske tidlig i arbeidet – før vi går igang med å skrive deklarasjoner og imperativer, men vanligvis etter at vi har funnet ut hvilke handlinger programmet skal kunne utføre – å bestemme hva slags objekter og hvilke objektmengder programmet skal holde styr på. Vi kan som regel få gode idéer til hvilke objekter og objektmengder programmet skal håndtere ved å se på den delen av den virkelige verden programmet skal modellere.

Noen eksempler:

Et medlemsregister: Anta vi skal lage et enkelt ordrestyrt program som skal brukes til å holde styr på data om medlemmer av en klubb. Programmet skal bl.a. kunne registrere data om nye medlemmer, melde ut medlemmer, og lage lister over alle registrerte medlemmer. En måte å lage dette programmet, er å la det håndtere et objekt for hvert medlem som registreres, og en objektmengde som har som medlemmer alle medlemsobjekter.

Et skoleprogram: Anta vi skal lage et kommandostyrt program som skal holde oversikt over ymse data om barna i en skole. Det kan da være fornuftig å lage programmet slik at det oppretter et objekt for hvert barn i skolen, dvs. objekter som inneholder data som beskriver enkeltbarn, og at det altså skal holde styr på slike "barneobjekter". Utfra en oversikt over hvilke kommandoer programmet skal kunne utføre (vi tar ikke med noen beskrivelse av disse kommandoene her), bestemmer vi at programmet for det første skal holde styr på en objektmengde som skal ha som medlemmer alle barneobjektene, dernest en objektmengde for hver klasse på skolen (den mengden skal selvfølgelig ha som medlemmer objektene til barna i klassen).

Et student/kurs-program: Et program som skal holde oversikt over data om studenter og kurs vil rimeligvis operere med objekter av to slag: Studentobjekter og kursobjekter. Gitt at programmet bl.a. skal holde rede på hvilke kurs hver student følger og også hvilke studenter som følger hvert kurs, bør det trolig holde styr på ihvertfall følgende objektmengder:

1. En objektmengde med alle studentobjektene som medlemmer.
2. En objektmengde med alle kursobjektene som medlemmer.
3. For hvert studentobjekt: En mengde som har som medlemmer alle kurs studenten følger.
4. For hvert kursobjekt: En mengde med alle studentene som følger kurset.

Et observasjonsprogram: Dette programmet skal holde styr på data om en del målestasjoner og observasjoner (av en eller flere størrelser) foretatt ved disse stasjonene. Programmet bør trolig bruke objekter av to slag: Stasjonsobjekter (ett slikt objekt for hver målestasjon) og observasjonsobjekter (ett for hver observasjon). Videre bør det holde rede på følgende objektmengder:

1. En mengde med alle stasjonsobjektene som medlemmer.
2. For hver stasjon en mengde som har som medlemmer et objekt for hver observasjon som er foretatt ved stasjonen.

2 Programmering av objektmengder

Etter at vi har bestemt seg for hva slags objekter og hvilke objektmengder et program skal holde styr på, og også hvilke handlinger programmet skal kunne utføre, er det gjerne på tide å definere (deklarere) en passende datastruktur med tilhørende operasjoner. Dersom vi bruker Simula som programmeringsspråk, vil en slik definisjon vanligvis bestå av:

- Deklarasjon av en klasse for hvert objektslag. En slik deklarasjon inneholder deklarasjoner av variable som utgjør den den indre datastrukturen til objekter av dette slaget. I tillegg inneholder klassen prosedyrer som gjør noe med eller observerer den indre datastrukturen.

- Deklarasjon av variabler som skal brukes til å holde styr på objektmengdene. Dette kan gjøres på mange ulike måter, noen av disse måtene beskrives i dette notatet.

- Deklarasjon av prosedyrer som brukes til å utføre nødvendige operasjoner på objektmengdene. Hvilke operasjoner som er nødvendige avhenger selvfølgelig av hvilke handlinger programmet skal kunne utføre, men det er vanlig at vi trenger prosedyrer for å utføre ihvertfall følgende operasjoner:

  1. *Melde inn* et nytt objekt i en objektmengde,
  2. *Finne fram* til et spesifisert medlem av en objektmengde (f.eks. finne et medlemsobjekt med et spesifisert navn),
  3. *Melde ut* (fjerne) et spesifisert medlem fra en objektmengde,
  4. *Traversere* en objektmengde, dvs. gå gjennom mengdens medlemmer, ett for ett i den rekkefølge de ligger i mengden, og gjøre noe med hvert enkelt medlem,
  5. *Observere* en objektmengde på ulike måter, f.eks. telle antall medlemmer i mengden eller finne gjennomsnittlig verdi på et eller annet dataelement som finnes hos alle objektene i mengden.

I IN105 omtales tre ganske ulike metoder for å håndtere objektmengder:

2.1 Array-metoden:

Her deklareres det et **ref**-array og en heltallsvariabel for hver objektmengde. Array'et skal brukes til referanser til objektene som er medlemmer i mengden, heltallsvariablen skal til enhver tid ha som verdi antall elementer i mengden. Dersom array-metoden brukes i skoleprogrammet beskrevet i det andre eksemplet ovenfor, vil programmet kunne inneholde f.eks. følgende deklarasjoner (vi forutsetter at det maksimalt er 500 elever på skolen, at det er seks klassetrinn hver med opp til fire parallelle klasser og at hver klasse har høyst 30 elever):

```
ref(Barn) array Skole(1 : 500),
                Klasse(1 : 6, 1 : 4, 1 : 30);
integer         antall_barn_i_skolen;
integer array   antall_barn_i_klasse(1 : 6, 1 : 4);
```

2.2 Pekerkjede-metoden:

Idéen i denne metoden er at for hver objektmengde deklareres det for det første en **ref**-variabel som skal ha som verdi en referanse til det første medlemmet i mengden. I tillegg sørger vi for (ved en passende deklarasjon i klassen som beskriver medlemsobjektene) at hvert objekt som kan være medlem av en mengde inneholder en **ref**-variabel som skal brukes til å inneholde en referanse til det neste objektet i mengden. Dersom et objekt kan være medlem av flere ulike objektmengder, må det inneholde en slik neste-peker for hver av de mulige objektmengdene det kan være med i.

Medlemsregister (forts.) Vi kan bruke pekerkjedemetoden i medlemsregisteret slik: For det første deklareres en variabel som til enhver tid skal inneholde en peker til det første medlemsobjektet:

```
ref(Medlem) forste_medlem;
```

For det andre tas det med en **ref**(Medlem)-variabel i deklarasjonen av klassen Medlem:

```
class Medlem;
  begin
    ref(Medlem) neste_medlem;
    <Andre deklarasjoner>;
  end;
```

Meningen er at variablen m . neste_medlem til enhver tid skal inneholde en peker til neste medlemsobjekt etter m. Dersom m er siste medlem, skal m . neste_medlem peke på **none**.

Skoleprogrammet (forts.) Pekerkjedemetoden kan brukes i skoleprogrammet ved å la det inneholde f.eks. følgende deklarasjoner:

```
ref(Barn)       forste_barn_i_skolen;
ref(Barn) array forste_barn_i_klasse(1 : 6, 1 : 4);
```

I tillegg utvides deklarasjonen av klassen Barn med følgende deklarasjoner:

```
class Barn;
  begin
    ref(Barn) neste_barn_i_skolen,
              neste_barn_i_klassen;
    <andre deklarasjoner>;
  end Barn;
```

2.3 Settools-metoden:

Settools er samling av ferdigdeklarerte klasser med verktøy som kan brukes til å holde styr på objektmengder. Resten av dette notatet beskriver den delen av dette verktøyet som er av interesse i IN105, og hvordan denne delen av verktøyet kan brukes.

3 Sett og sekvenser

Settools inneholder verktøy for å håndtere objektmengder av flere slag. To av disse mengdeslagene er av interesse i IN105:

- Sekvenser
- Sett

Objekter som skal kunne være medlemmer av et sett må være *identifiserbare*, dvs. at det må finnes en såkalt *identifikasjonsnøkkel* – vanligvis et navn, fødselsnummer, identifikasjonsnummer eller lignende – som er slik at alle medlemmene av settet har ulike nøkkelverdier. Objekter som skal være medlemmer av en sekvens trenger ikke å være identifiserbare, de trenger altså ikke ha noe entydig navn eller lignende.

Vi kan si at et *sett* er en mengde av objekter som har entydig identifikasjonsnøkkel, mens en *sekvens* er en mengde av objekter som ikke trenger ha noen slik nøkkel. Videre er det slik at i et sett ligger medlemmene alfabetisk sortert etter stigende nøkkelverdi, mens de i sekvenser ligger i innmeldingsrekkefølge.

Dersom et program skal holde styr på en objektmengde der medlemsobjektene ikke er identifiserbare, dvs. ikke har noe navn eller annet som er slik at to medlemmer alltid har ulik verdi, må vi la objektmengden være en sekvens. Dersom imidlertid hvert objekt har en identifikasjonsnøkkel, slik at objektene er identifiserbare, kan vi la objektmengden være et sett.

En viktig fordel sett har framfor sekvenser, er at identifikasjonsnøkler kan brukes til å spesifisere hvilket medlem av et sett vi vil gjøre noe med, f.eks. se på eller melde ut av settet. I alle sett finnes verktøy som kan brukes til å finne fram et medlem som har en bestemt nøkkel. Dette betyr at dersom vi kjenner nøkkelen til et bestemt medlem av et sett, kan vi "spørre" settet om å finne fram dette medlemmet, og dermed ganske direkte få tak i det. Skal vi ha tak i et bestemt medlem av en sekvens, må vi gå sekvensielt gjennom alle medlemmene av sekvensen, ett for ett, inntil vi finner det vi leter etter.

Noen eksempler som viser bruk av sett og sekvenser:

Medlemsregister (forts.) Vi antar at medlemmene av klubben har entydige navn, som kan brukes som identifikasjons-nøkler. Vi kan representere klubben som et sett, som da til enhver tid skal inneholde pekere til medlems-objekter for alle registrerte medlemmer av klubben.

Skoleprogrammet (forts.): Dersom vi antar at alle barn på skolen har entydig navn, kan vi bruke navn som identifikasjonsnøkkel i skolemengden og klassemengdene. Det betyr at vi kan la disse mengdene være sett eller sekvenser. Vi velger å bruke sett i alle tilfeller. Årsaken til at vi ikke lar mengdene være sekvenser, er at flere av de kommandoene vi kan anta programmet skal kunne utføre, er slik at vi skal finne fram til barn med bestemte navn (enten for å se på eller endre data som er lagret om barnet, eller for fjerne alt som er lagret om barnet), og da er det fordelaktig å bruke sett.

At barn har entydige navn er selvfølgelig ganske urealistisk. I et et mer realistisk program bør vi derfor bruke noe annet enn navn som identifikasjonsnøkkel, f.eks. personnummer .

Student/kurs-programmet (forts.): Personnummer kan brukes som nøkkel for studenter, kurskode (f.eks. IN105) som nøkkel for kursene. Vi lar alle mengder være sett.

Observasjonsprogrammet (forts.): Det er rimlig å anta at målestasjonene har entydige navn som kan brukes som nøkkel. Mengden av alle målestasjons-objekter lar vi være et sett. Observasjonene kan vi anta ikke er identifiserbare, hvilket betyr at mengden som består av alle observasjoner ved en stasjon må være en sekvens. I noen situasjoner kan det selvfølgelig tenkes at det kan lages entydige nøkler for observasjonene, f.eks. kan kanskje tidspunktet brukes (sett sammen år, måned, dag, time, minutt og sekund til en nøkkel), eller kanskje finnes det løpenummere i observasjonsprotokollene som identifiserer observasjonene. I slike situasjoner kan vi selvfølgelig gjerne la mengden av observasjoner foretatt ved en stasjon være et sett.

4 Hovedstruktur i Settools

De tre viktigste Settools-klassene er element, set og sequence. Disse klassene er deklarert som følger (en god del forenklet her):

```
class element;
    virtual: text procedure key;
  begin
    <Deklarasjoner av noen prosedyrer
      (se avsnitt 6.1 nedenfor) >;
  end element;
element class set;
  begin
    <Deklarasjoner av verkty for  hndtere
      sett (se avsnitt 6.2 og 6.3) >;
  end set;
element class sequence;
  begin
    <Deklarasjoner av verkty for  hndtere
      sekvenser (se avsnitt 6.2 og 6.4) >;
  end sequence;
```

Ethvert objekt som skal være medlem av et sett eller en sekvens, må være en instans av klassen element eller av en subklasse av element. Det betyr at en klasse som beskriver objekter som skal kunne være medlemmer av sett eller sekvenser, må deklareres som en subklasse av element. Eksempler gis nedenfor. Legg merke til at klassene set og sequence er subklasser av klassen element. Hensikten med det er at sett og sekvenser selv kan være medlemmer i sett og sekvenser.

Dersom et objekt skal være medlem av et sett må klassen som objektet er en instans av, inneholde deklarasjon av en **text**-prosedyre med navn key som for ethvert objekt skal ha som verdi objektets entydige identifikasjonsnøkkel. Dette er ikke nødvendig dersom objektet bare skal være medlem i sekvenser og aldri i sett.

Noen eksempler:

Medlemsregister (forts.) En datastruktur for medlemsregisteret kan deklareres slik:

```
element class Medlem(navn); text navn;
  begin
    text procedure key; key :- navn;
    <Deklarasjoner av evt. andre attributter >;
  end Medlem;
ref(Set) Klubben;
```

Skoleprogrammet (forts.): En datastruktur for skoleprogrammet kan f.eks. deklareres slik dersom Settools benyttes:

```
element class Barn;
  begin
    text navn;
    text procedure key; key :- navn;
    <Deklarasjoner av andre attributter >;
  end Barn;
ref(Set) Skole;
ref(Set) array Klasse(1 : 6, 1 : 4);
```

Student/kurs-programmet (forts.): En datastruktur for et program som skal holde styr på data om studenter og kurs kan f.eks. deklareres som følger:

```
element class Student(persnr); text persnr;
  begin
    ref(Set) kurs_liste;
    text procedure key; key :- persnr;
    <Deklarasjoner av andre attributter >;
    kurs_liste :- new Set;
  end Student;
element class Kurs(kode); text kode;
  begin
    ref(Set) studenter;
    text procedure key; key :- kode;
    <Deklarasjoner av andre attributter >;
    studenter :- new Set;
  end Kurs;
ref(Set) Alle_kurs, Alle_studenter;
```

Observasjons-programmet (forts.): Et program som skal holde styr på data om observasjoner foretatt på diverse målestasjoner kan ha følgende datastruktur:

```
element class Stasjon;
  begin
    text stasjons_navn;
    ref(sequence) observasjoner;
    text procedure key; key :- stasjons_navn;
    <Deklarasjoner av andre attributter >;
    observasjoner :- new sequence;
```

```
    end Stasjon;
  element class Observasjon;
    begin
      <Deklarasjoner av data-attributter >;
      <Deklarasjoner av prosedyrer >;
    end Observasjon;
  ref(Set) Stasjoner;
```

5 Hvordan ta **Settools** i bruk i et program

Deklarasjoner av alt Settools-verktøyet finnes på filen

```
~in105/lib/settools . sim
```

Denne filen har følgende innhold (noe forenklet):

```
class settools; <...>;
class element; <...>;
element class Container; <...>;
Container class Set; <...>;
Container class Sequence; <...>;
<Diverse andre klasser (utenfor IN105)>;
```

Som man ser er dette noe annerledes enn beskrevet i begynnelsen av avsnitt 4 ovenfor. Her er det to klasser som ikke er omtalt tidligere: settools og Container. Den første av disse forklares nedenfor. Den andre, altså Container, er strengt tatt unødvendig, og er bare tatt med i den foreliggende versjonen av Settools for å redusere den samlede størrelsen på deklarasjonene av klassene Set og Sequence: Klassen Container inneholder nemlig alt som er felles for disse to klassene (og det er det meste). Men Container-klassen kan ikke brukes alene, til det er den for svak.

Den greieste måten å bruke Settools i et program er å deklarere settools som en ekstern klasse. Det kan gjøres slik:

```
begin
  external class settools;
  <Bruk av klassene  element, Set, Sequence.
    (Eksempler gis nedenfor.) >
end;
```

For å kompilere et program som bruker eksterne klasser fra ~in105-området (f.eks. alle klasser på filen settools . sim), må man bruke følgende kompileringskommando:

```
sc  -in105  prog
```

(eventuelt "sc -in105w prog" dersom programmet også bruker Windowtools). Dette er forklart i notatet ~in105/external . DVI.

Selve klassen settools (som ligger først på filen ~in105/lib/settools . sim) inneholder ingenting av interesse. Den eneste årsak til at denne nesten tomme klassen er tatt med, er å gjøre det enklest mulig å få tak i alle klassene på filen settools . sim. Det er nemlig slik (ved den Simula-kompilatoren som for tiden brukes ved våre undervisningsmaskiner) at dersom en fil inneholder flere moduler (klasser og/eller prosedyrer), og den første av disse har samme navn som filen (bortsett fra ".sim"-delen av filnavnet), kan man gjøre alle modulene på filen tilgjengelige i et program ved å deklarere denne første modulen som ekstern (som vist ovenfor).

6 De viktigste **Settools** klassene

6.1 Klassen **element**

Klassen element er deklarert som følger (stadig noe forenklet):

```
class element;
    virtual: text procedure key;
  begin
    text procedure key; key :- notext;
  end of element;
```

Virkningen av at prosedyren key er spesifisert til å være virtuell, er at vi kan deklarere egne versjoner av denne prosedyren i subklasser av element.

6.2 Klassen **Container**

Klassen Container inneholder deklarasjoner av prosedyrer (og noe annet) som er felles for sett og sekvenser. De viktigste (i IN105-sammenheng) av disse prosedyrene er for det første prosedyrer for å finne størrelsen på en objektmengde (size og is_empty), for det andre prosedyrer som kan brukes til å gå igjennom alle medlemmene av en mengde ett for ett (first_element og next_element), og for det tredje en prosedyre som kan brukes til å utføre bestemte handlinger for alle medlemmene av en mengde (for_each_element).

Container er for svak til å benyttes alene, til det har klassen for få verktøy (den viktigste mangelen er at klassen ikke har verktøy for å melde inn nye medlemmer). Det er derfor ikke meningen at et program som bruker Settools skal benytte Container direkte (men indirekte brukes klassen når man bruker klassene Set eller Sequence, som begge arver alt som er deklarert i Container).

Container er deklarert som følger (skissemessig og forenklet):

```
class Container;
  begin
    integer procedure size;
      size := <antall medlemmer>;

    Boolean procedure is_empty;
      is_empty := size = 0;

    ref(element) procedure first_element;
      first_element :- <frste medlem>;

    ref(element) procedure next_element;
      next_element :- <neste medlem>;

    ref(Container) procedure for_each_element(p);
        procedure p;
      begin
        <Utfr p(el) for hvert medlem el>;
        for_each_element :- this Container;
      end;

    ref(Container) procedure make_empty;
      begin
        <Meld ut alle medlemmer>;
        make_empty :- this Container;
      end;

    procedure remove_current;
      <Meld ut det medlemmet som sist ble vist til
        av first_element eller next_element>;

    ref(element) procedure last_element;
      last_element :- <siste medlem>;

    ref(element) procedure element_number(elnr);
        integer elnr;
      element_number :- <medlem nr. elnr>;

    ref(Container) procedure map(f);
        ref(element) procedure f;
      <Erstatt ethvert medlem el med f(el)>;

    ref(Container) procedure filter(b);
        Boolean procedure b;
      <Meld ut hvert medlem som ikke oppfyller b,
        dvs. hvert medlem el hvor b(el) er false>;

  end of Container;
```

Prosedyrene make_empty, for_each_element, map og filter er deklarert som **ref**(Container)-funksjoner. De får alle som verdi en referanse til sin egen Container. Dette oppnås ved at det siste som utføres i alle disse prosedyrene er følgende tilordning:

```
<funksjons-navn>:- this Container;
```

Hensikten med at disse fire prosedyrene får en slik verdi kan best ses av eksempler, men er av liten eller ingen interesse i IN105. De to prosedyrene map og filter er generelt av liten eller ingen interesse i IN105 (ihvertfall foreløpig).

6.3 Klassen Set

Klassen Set inneholder for det første alle prosedyrene som arves fra Container, for det andre en prosedyre for å melde inn nye medlemmer (add_element), for det tredje en prosedyre for å avgjøre om et sett har et medlem med en bestemt nøkkel (has_element), for det fjerde prosedyrer å finne og fjerne medlemmer med bestemte nøkler (find_element, find_element_after og remove_element, og for det femte en prosedyre for å sette sammen to sett til ett (union).

Klassen Set er deklarert som følger (skissemessig):

```
Container class Set;
  begin
    ref(Set) procedure add_element(el);
        ref(element) el;
      begin
        <Meld inn el som nytt medlem. Dersom det
          finnes et medlem fra fr med sammme
          nkkel som el, meldes det gamle
          medlemmet ut;>
        add_element :- this Set;
      end;

    ref(element) procedure find_element(key);
        text key;
      find_element :- if <det fins et medlem el hvor
                            el . key = key >
                        then el
                        else  none;

    Boolean procedure has_element(key); text key;
      has_element := find_element(key) =/= none;

    ref(element) procedure find_element_after(key);
        text key;
      find_element :- if <det fins et medlem el hvor
                            el . key >= key >
                        then <frste slike medlem>
                        else  none;

    ref(element) procedure remove_element(key);
        text key;
      if <det fins et medlem el hvor el . key = key >
        then begin
               <Meld el ut av settet>;
               remove_element :- el;
             end
        else  remove_element :- none;

    ref(Set) procedure union(s); ref(Set) s;
      begin
```

```
        <Meld alle medlemmer av s inn i
          dette Set'tet>;
        union :- this Set;
      end;
  end of set;
```

Prosedyren som brukes til å melde nye medlemmer inn i et sett, altså add_element, er deklarert som en **ref**(Set)-prosedyre, og får som verdi en peker til sitt sett. Hensikten med dette er at det blir lett å skrive imperativer som melder flere medlemmer inn. For eksempel kan vi skrive:

```
    et_sett . add_element(el1)
            . add_element(el2)
            . add_element(el2);
```

6.4 Klassen **Sequence**

I tillegg til arv fra Container, inneholder Sequence en prosedyre for å melde inn nye medlemmer (append), og en for å sette sammen to sekvenser til én (concatenate). Sequence er deklarert som følger (skissemessig):

```
Container class Sequence;
  begin
    ref(Sequence) procedure append(el);
        ref(element) el;
      <Meld el inn (plasseres sist)>;
    ref(Sequence) procedure concatenate(s);
        ref(Sequence) s;
      begin
        <Meld inn alle medlemmer av s
          i denne sekvensen>;
        concatenate :- this Sequence;
      end;
  end of Sequence;
```

7 Noen eksempler

7.1 Et medlemsregister

Anta vi skal lage et program som skal brukes til å holde styr på enkle data om medlemmer av en klubb. Programmets spesifikasjoner sier at det skal være ordrestyrt og istand til å utføre ordre for å

- Legge inn data om et nytt medlem,
- Fjerne alle data om et medlem (utmelding),
- Lage en liste over alle medlemmene,
- Skrive alle data til fil,
- Lese data fra fil,
- Avslutte.

For å forenkle beskrivelsen her, antar vi at det eneste som registreres av data om hvert medlem, er medlemmets navn (som vi antar er entydig). Det er imidlertid ikke vanskelig å utvide programmet slik at det holder styr på mer data om medlemmene.

7.1.1 Datastruktur

Programmet skal holde håndtere objekter av ett slag, nemlig medlemsobjekter. Det skal videre holde styr på én objektmenngde, nemlig mengden av medlemsobjekter. En enkel Settools-basert datastruktur for dette, kan deklareres slik:

```
element class Medlem(navn); text navn;
  begin
    text procedure key; key :- navn;
  end;

ref(Set) Klubben;
```

7.1.2 Melde medlemmer inn og ut

En prosedyre for å melde et nytt medlem inn i klubben kan deklareres som følger:

```
procedure NyttMedlem;
  begin text navn;
    navn :- pt . ask_for_text("Navn? ");
    if Klubben . has_element(navn)
      then outline(navn & " er i bruk")
      else  Klubben . add_element(
                      new Medlem(navn));
  end;
```

En prosedyre for å melde et navngitt medlem ut av klubben kan programmeres slik:

```
procedure Utmelding;
  begin text navn; ref(Medlem) m;
    navn :- pt . ask_for_text("Navn? ");
    m :- Klubben . remove_element(navn);
    outline(if m == none
      then "Ingen heter " & navn
      else "Medlem " & m . key &
           " er utmeldt");
  end;
```

7.1.3 Medlemslister

Skal det lages en liste med navnene på alle klubbens medlemmer, må programmet gå sekvensielt igjennom alle elementene i settet Klubben. En slik gjennomgang kan programmeres på to ganske ulike måter. Den første er ved bruk av prosedyrene first_element og next_element, f.eks. slik:

```
procedure MedlemsListe;
  begin ref(Medlem) m;
    outtext("Medlems-liste: ");
    m :- Klubben . first_element;
    while m =/= none do
      begin
        outline("     " & m . navn);
        m :- Klubben . next_element;
      end;
    outimage;
  end;
```

Alternativt kan det samme programmeres ved bruk av prosedyren for_each_element:

```
procedure MedlemsListe;
  begin

    procedure skriv_navn(m); ref(Medlem) m;
      outtext("   " & m . navn);

    outline("Medlems-liste: ");
    Klubben . for_each_element(skriv_navn);
    outimage;

  end;
```

Disse to måtene å få utført en bestemt handling for alle medlemmene av et sett, er helt ekvivalente. Hvilken vi velger, er nesten en smaksak. Det er nok slik at de fleste synes den første måten er enklest å forstå og å benytte, ihvertfall i begynnelsen. Men dersom man tar seg bryet med å lære seg å bruke den andre måten, vil man nok se at den vanligvis fører til kortere programmer. Det vil faktisk ofte være lettere å se strukturen i programmer som bruker den andre måten – ihvertfall når man har vennet seg til en slik mer *funksjonell* måte å programmere på.

7.1.4 Til og fra fil

En prosedyre for å skrive alle data til fil, kan programmeres slik:

```
procedure Tilfil;
  inspect new outfile(<...>) do
    begin ref(Medlem) m;
      open(blanks(50));
      m :- Klubben . first_element;
      while m =/= none do
        begin
          m . til_fil(this outfile);
          m :- Klubben . next_element;
        end;
      close;
    end;
```

Klassen Medlem må da inneholde en deklarasjon av en prosedyre til_fil, f.eks. slik:

```
procedure til_fil(fil); ref(outfile) fil;
  begin fil . outtext(navn); fil . outimage end;
```

Alternativt kan prosedyren Tilfil programmeres slik:

```
procedure Tilfil;
  inspect new outfile(<...>) do
    begin

      procedure til_fil(m); ref(Medlem) m;
        m . til_fil(this outfile);

      open(blanks(50));
      Klubben . for_each_element(til_fil);
      close;

    end;
```

En prosedyre som leser fra fil det prosedyren Tilfil har skrevet, kan programmeres slik:

```
procedure Frafil;
  inspect new infile(<...>) do
    begin
      open(blanks(50));
      inimage;
      while not endfile do
        begin
          Klubben . add_element(
                new Medlem(intext(50) . strip));
          inimage;
        end;
      close;
    end;
```

7.1.5 Hele programmet

Hele programmet beskrevet i dette avsnittet finnes på filen `~in105/klubb.sim`.

7.2 Kurs-påmelding

Anta vi skal lage et program som holde styr på data om visse kurs og elever, spesielt om hvilke kurs hver av elevene er påmeldt, og hvilke elever som er påmeldt hvert av kursene. Programmet skal være ordrestyrt, og skal kunne utføre ihvertfall følgende ordre:

- Registrere data om et nytt kurs,
- Registrere data om en ny elev,
- Melde en spesifisert elev på et spesifisert kurs,
- Melde en spesifisert elev av et spesifisert kurs,
- Melde en spesifisert elev av alt,
- Avlyse et kurs,
- Lage en liste over alle elever som er påmeldt et spesifisert kurs,
- Lage en liste over alle kurs som en spesifisert elev er påmeldt,
- Lage en liste over alle kurs,
- Lage en liste over alle elever,
- Avslutte.

Vi antar at hvert kurs er identifisert ved en entydig kode, og at hver elev har et entydig navn. For enkelthets skyld antar vi at det ikke registreres andre faste data om hvert kurs enn kursets kode, og ikke andre faste data om elevene enn deres navn.

7.2.1 Datastruktur

Programmet skal holde håndtere to slags objekter: elever og kurs. Det skal styr på ihvertfall følgende objektmengder:

- En med alle elever som medlemmer,
- En med alle kurs som medlemmer,
- For hver elev: alle kurs eleven er påmeldt,
- For hvert kurs: alle elever som er påmeldt kurset,

Brukes Settools; kan en passende datastruktur deklareres f.eks. slik:

```
element class Elev(navn); text navn;
  begin
    text procedure key; key :- navn;
    ref(Set) KursListe;
    <Deklarasjon av prosedyrer (se nedenfor)>;
    KursListe :- new Set;
  end;
element class Kurs(kode); text kode;
  begin
    text procedure key; key :- kode;
    ref(Set) ElevListe;
    <Deklarasjon av prosedyrer (se nedenfor)>;
    ElevListe :- new Set;
  end;
ref(Set) AlleElever, AlleKurs;
```

7.2.2 Nye elever og kurs

En prosedyre som kan brukes til å registrere en ny elev kan programmeres slik:

```
procedure NyElev;
  begin text navn;
    navn :- pt . ask_for_text("Elevens navn? ");
    if AlleElever . has_element(navn)
      then outline(navn & " er i bruk!")
      else AlleElever . add_element(
                        new Elev(navn));
  end;
```

En prosedyre for registrering av nye kurs, kan deklareres helt tilsvarende.

7.2.3 Kurs-påmelding

En prosedyre som kan brukes til å registrere en kurspåmelding kan programmeres slik:

```
procedure Paamelding;
  begin text navn, kode; ref(Elev) e; ref(Kurs) k;
    navn :- pt . ask_for_text("Elevens navn? ");
    e :- AlleElever . find_element(navn);
    if e == none
    then outline("Ingen elev " & navn)
    else begin
        kode :- pt . ask_for_text("Kurs kode? ");
        k :- AlleKurs . find_element(kode);
        if k == none
        then outline("Intet kurs " & kode)
        else begin
            e . Paameld_kurs(k);
            k . Paameld_elev(e);
            outline(navn & " er påmeldt " &
                  kode);
          end;
      end;
  end;
```

Her forutsettes det at meldingen Paameld_kurs(k) kan sendes til en elev e, og meldingen Paameld_elev(e) til et kurs k. Vi må derfor deklarere en prosedyre Paameld_elev i klassen Kurs og en prosedyre Paameld_kurs i klassen Elev. Den første av disse kan deklareres slik:

```
element class Kurs(kode); text kode;
  begin
    <...>;
```

```
      ref(Set) ElevListe;
      <...>;
      procedure Paameld_elev(e); ref(Elev) e;
         ElevListe . add_element(e);
      <...>;
   end Kurs;
```

Prosedyren Paameld_kurs kan deklareres ganske likt i klassen Elev.

7.2.4 Kurs-avmelding

En prosedyre som kan brukes til å melde en elev av et kurs, kan programmeres slik:

```
procedure Avmelding;
   begin text navn, kode; ref(Elev) e; ref(Kurs) k;
      navn :- pt . ask_for_text("Elevens navn? ");
      e :- AlleElever . find_element(navn);
      if e == none
      then outline("Ingen elev " & navn)
      else begin
            kode :- pt . ask_for_text("Kurs kode? ");
            k :- e . finn_kurs(kode);
            if k == none
            then outline(navn & " er ikke påmeldt "
                         & kode)
            else begin
                  e . Avmeld_fra_kurs(kode);
                  k . Avmeld_elev(navn);
               end;
         end;
   end;
```

Prosedyrene finn_kurs og Avmeld_fra_kurs som brukes her, kan deklareres som følger (inne i klassen Elev):

```
ref(Kurs) procedure finn_kurs(kode); text kode;
   finn_kurs :- KursListe . find_element(kode);
procedure Avmeld_fra_kurs(kode); text kode;
   KursListe . remove_element(kode);
```

Prosedyren Avmeld_elev (som også brukes i Avmelding) kan deklareres helt tilsvarende i klassen Kurs.

7.2.5 Avlyse et kurs

Prosedyren nedenfor kan brukes til å avlyse et kurs. Vær klar over at dette ikke kan gjøres bare ved å fjerne kurset fra settet AlleKurs. I tillegg må nemlig også kurset fjernes fra kurslistene til de elevene som måtte være påmeldt kurset.

```
procedure KursAvlysning;
   begin text kode; ref(Kurs) k;
      kode :- pt . ask_for_text("Kode? ");
      k :- AlleKurs . remove_element(kode);
      if k == none
         then outline("Kurs-kode " & kode &
                      " er ukjent!")
         else begin
               k . Avmeld_alle_elever;
               outline("Kurs " & kode &
                       " er avlyst");
            end;
   end;
```

Prosedyren Avmeld_alle_elever som brukes her, kan deklareres slik i klassen Kurs:

```
procedure Avmeld_alle_elever;
   begin ref(Elev) e;
      e :- ElevListe . first_element;
      while e =/= none do
         begin
            e . Avmeld_fra_kurs(kode);
            e :- ElevListe . next_element;
         end;
      ElevListe . make_empty;
   end;
```

Alternativt:

```
procedure Avmeld_alle_elever;
   begin

      procedure avmeld(e); ref(Elev) e;
            e . Avmeld_fra_kurs(kode);

      ElevListe . for_each_element(avmeld);
      ElevListe . make_empty;

   end;
```

7.2.6 Melde en elev av alle kurs

Følgende prosedyre kan brukes til å melde en elev av alle kurs han eller hun måtte være påmeldt:

```
procedure ElevSlutt;
   begin text navn; ref(Elev) e;
      navn :- pt . ask_for_text("Navn? ");
      e :- AlleElever . remove_element(navn);
      if e == none
         then outline("Ingen elev heter " & navn)
         else begin
               e . Avmeld_fra_alle_kurs;
               outline("Elev " & navn &
                       " er sluttet");
            end;
   end;
```

Prosedyren Avmeld_fra_alle_kurs kan deklareres ganske tilsvarende til Avmeld_alle_elever fra avsnitt 7.2.5.

7.2.7 Lister

Følgende prosedyre kan brukes til å lage en liste over alle elever. For hver elev skrives kodene til alle kurs eleven måtte være påmeldt.

```
procedure SkrivElevListe;
  begin
    procedure skriv(e); ref(Elev) e;
      e . VisKursListe;

    AlleElever . for_each_element(skriv);

  end;
```

Prosedyren VisKursListe som brukes her kan deklareres slik i klassen Elev:

```
procedure VisKursListe;
  begin
    procedure skriv_kode(k); ref(Kurs) k;
      outtext(" " & k . kode);

    outtext(navn & " deltar på følgende " &
            tt . int_as_text(KursListe . size) &
            " kurs:");
    KursListe . for_each_element(skriv_kode);
    outimage;

  end;
```

7.2.8 Hele programmet

Hele programmet beskrevet i dette avsnittet finnes på filen `~in105/kurs . sim`.

7.3 Et observasjonsprogram

Filen `~in105/observ.sim` inneholder et program som holder styr på data om observasjoner. Programmet håndterer to slags objekter: *observasjoner* og *stasjoner*, og inneholder derfor deklarasjoner av følgende to klasser:

```
element class Observasjon;
  begin
    ref(Dato) obs_dato;
    real verdi;

    <Deklarasjoner av prosedyrer bl.a. for
          lese og skrive observasjonsdata>;

  end Observasjon;

element class Stasjon(navn); text navn;
  begin
    text procedure key; key :- navn;

    ref(sequence) observasjoner;

    <Deklarasjoner av diverse prosedyrer>;

    observasjoner :- new sequence;

  end Stasjon;
```

Klassen Dato som benyttes i klassen Observasjon inneholder prosedyrer for å holde styr på datoer (dag, måned og år).

En av prosedyrene i klassen Stasjon kan brukes til å fjerne alle observasjoner som er eldre enn en gitt dato. Den kan deklareres slik:

```
element class Stasjon(navn); text navn;
  begin
    . . .
    procedure Fjerngamleobs(gd); ref(Dato) gd;
      begin ref(Observasjon) obs;
        obs :- observasjoner . first_element;
        while obs =/= none do
          begin
            if obs . obs_dato . eldre_enn(gd)
              then observasjoner . remove_current;
            obs :- observasjoner . next_element;
          end;
      end;
    . . .
  end Stasjon;
```

Legg merke til bruken av prosedyren remove_current her: Denne prosedyren er deklarert i klassen Container og er derfor tilgjengelig i Set og Sequence (og andre subklasser av Container). Virkningen av å invokere prosedyren i en sekvens (eller et sett) er at det elementet som sist ble gitt som verdi av enten first_element eller next_element blir fjernet fra sekvensen (settet).

7.4 Andre eksempler

Filene

- `~in105/elev2.sim`
- `~in105/bolig.sim`
- `~in105/julegave.sim`

inneholder fullstendige eksempler på bruk av Settools.

8 Implementasjon av Settools

Filen `~in105/settools.sim` inneholder en full_stendig deklarasjonen av alt verktøyet i Settools. Akkurat som man ikke trenger å forstå innmaten av en radio for å kunne bruke den, er det ikke på noen må nødvendig å forstå detaljene i implementasjonen av Settools for å kunne bruke verktøyet i Settools. Ta gjerne en titt på filen `~in105/settools.sim`, men gjør ikke noe arbeidskrevende forsøk på å forstå alt sammen (dersom du da ikke er spesielt interessert eller veldig nysgjerrig).