En note om Programmering

Kurt Nørmark
Institut for Datalogi
Aalborg Universitet
normark@cs.aau.dk

Resumé

Denne note er en introduktion til programmering. Formålet er at give dig et indblik i hvad programmering egentlig er for noget. Jeg vil vise at programmering kan foregå på forskellige måder, og at der er mange forskellige udfordringer forbundet med at programmere. Noten vil ikke knytte sig til et bestemt programmeringssprog. Noten vil kunne supplere et egentlig undervisningsmateriale, der støtter dig i en bestemt form for programmering i et udvalgt programmeringssprog.

1 Introduktion

Din computer består af hardware og software. Hardware er elektronik - software er programmer og data. Programmerne er de vigtigste dele i din computer - og programmering er en central aktivitet i faget Informationsteknologi. Uden mulighed for at programmere en computer vil næsten alle andre aspekter af informationsteknologi falde bort og miste deres betydning.

Men hvad søren er programmering for noget, vil du måske spørge. Vi giver dig et muligt svar i denne note. Når du kommer til den sidste side i noten er det målet, at du forstår en hel del mere om programmering.

Vi starter med at se på et konkret programmeringsproblem, som vil være en rød tråd igennem hele noten. Det første bud på en løsning vil blive programmeret "på en klassisk måde", i et såkaldt imperativt programmeringssprog. Efter dette program vil vi tale om programstruktur, og om vigtigheden ved af at løsrive sig fra nogle af detaljerne i programmet gennem abstraktion. Dernæst kommer et afsnit

om fejl og test. I den sidste del af noten ser vi på løsninger af det gennemgående problem i både objekt-orienteret programmering og i funktionsprogrammering. Noten sluttes af med et forsøg på at forstå programmørens arbejde i forhold til andre former for arbejde.

2 Problemløsning og programmering

I bred forstand handler programmering om at at løse et problem som er relateret til håndtering af information. Næsten alle sådanne problemer løses i dag ved brug af en computer. Som et eksempel vil vi starte med at se på et program, der løser følgende problem:

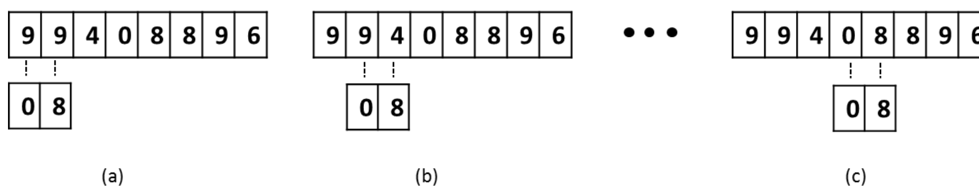
> Du leder efter et telefonnummer i din telefonbog, hvor du kun husker en enkelt del af nummeret. Du ønsker f.eks. at finde den eller de numre der indeholder tallet 234 et sted i midten, et nummer som begynder med 98, eller et nummer der afsluttes med 7913.

Vi kan naturligvis være så heldige at problemet allerede er løst i et eller andet program vi har kørende på vores computer, eller som kan downloades fra nettet. Ofte er problemet dog så specielt, at det ikke allerede er løst. Eller også passer et eksisterende program ikke ind i den sammenhæng, hvor løsningen skal anvendes. Vi skriver derfor programmet selv!

2.1 Opdel et problem i delproblemer

Når vi beslutter at skrive et program, der løser et problem, bør vi altid dele problemet op i et antal delproblemer. Det er fordi problemet typisk er for kompliceret - og for stort - til at det kan løses "i et hug". Sådan er det også med telefonnummer problemet. Her er to mulige delproblemer:

1. For ét bestemt telefonnummer ønsker vi at finde ud af om - og hvor - det indeholder et bestemt delnummer (en bestemt rækkefølge af sammenhængende cifre).
2. For hele telefonbogen ønsker vi at finde de numre, som indeholder et bestemt delnummer.



Figur 1: *Tre forskellige situationer i vores eftersøgning af delnummeret 08 i telefonnummeret 99408896.*

Det er ret oplagt, at hvis vi har løst det første delproblem kan vi løse det andet problem på følgende måde:[1]

```
1 FOREACH telefonnummer IN telefonBog
2   LET resultat = FindNummer(telefonnummer, delnummer)
3     IF resultat
4     THEN udskrivResultat(telefonnummer, delnummer, resultat)
```

I første linje får vi alle telefonnumre i hånden én for én - vi gennemløber altså alle telefonnumre i telefonbogen. I anden linje finder vi ud af om delnummeret findes i telefonnummeret. Her udnytter vi løsningen på det første delproblem, som vi dog først løser om lidt. Hvis vi får et brugbart resultat tilbage fra denne del af problemløsningen udskrives resultatet i linje 4. Resultatet vil fortælle os om delnummeret findes i telefonnummeret, og om delnummeret er placeret først, sidst eller i midten.

Det første delproblem er en anelse vanskeligere at løse. Med udgangspunkt i et eksempel har vi skitseret en mulig løsning i figur 1, hvor vi leder efter 08 i telefonnummeret 99408896. Først tester vi om 08 findes i starten af telefonnummeret. Da det ikke er tilfældet glider sammenligningen gradvis mod højre. I hver runde skal vi være indstillet på at sammenligne alle cifre i 08 med en del af telefonnummeret. I figur 1(c) ser vi at det lykkes at finde 08 i sammenligning nummer fire.

[1] Jeg viser forholdvis løse skitser af programmer, som ikke er skrevet i noget bestemt programmeringssprog. Jeg forventer ikke du umiddelbart forstår disse, men jeg håber at du gradvist vil forstå flere og flere dele af programmerne, når jeg forklarer deres betydning.

2.2 Se efter gentagelser

Som mennesker vi kan forholdsvis let overskue at delnummeret 08 findes i fjerde position i det givne telefonnummer. Det kan en computer ikke. Når vi programmerer løsningen skal vi gå systematisk til værks i små skridt. En computer kan umiddelbart kun sammenligne to cifre med hinanden i et skridt. Udfordringen i vores program bliver at programmere passende *gentagelser* af ciffer sammenligninger, som ender op med at løse det første delproblem. Der kan samlet set blive tale om mange sammenligninger af cifre - men det gør ikke noget da computeren er lynhurtig til at foretage sådanne beregninger. Computeren kan sammenligne millioner af cifre hvert sekund. Computerens hastighed - som giver mulighed for at gentage små beregninger mange gange - er den store styrke som i dette eksempel afhjælper computerens mangel på "menneskeligt overblik".

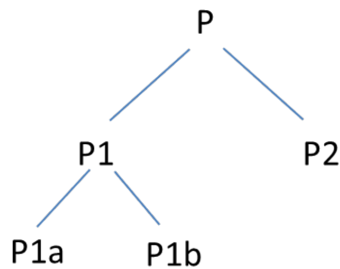
I en typisk løsning på det første delproblem vil der være to *gentagelser* inden i hinanden (to indlejrede løkker i programmet). Den yderste sørger for at delnummeret (08) i starten placeres helt til venstre, og at det i hvert gennemløb bringes et skridt længere mod højre (se figur 1). I den inderste gentagelse sammenlignes cifre parvis, om nødvendigt lige så mange gange som der er cifre i delnummeret. Dette skitseres i følgende program:

```
1  WHILE (delnummer ikke placeret til hoejre i telefonnummer)
2    WHILE (ciffer i delnummer matcher ciffer i telefonnummer)
3      tael ciffernummer op
4
5    IF (telefonnummer er udtoemt OG delnummer er ikke fundet)
6    THEN resultatet er: delnummer blev ikke fundet
7    ELSE IF (delnummer er fundet)
8    THEN resultatet er: placeringen af delnummer i telefonnummer
9    ELSE tael placeringen op
```

Se godt på programmet, og forsøg på at forstå det i forhold til figur 1. Den *placering* der bliver nævnt svarer til placeringen af starten af delnummeret i telefonnummeret, som vist i figur 1. Tilsvarende udpeger *ciffernummer* det ciffer der sammenlignes i delnummeret og telefonnummeret.

Ud over gentagelser ser vi også eksempler på *forgreninger* i linjerne 5-9 i programmet. Forgreninger kan programmeres med if-then-else. Betingelsen efter ordet if afgør om then-delen eller else-delen bliver kørt.

Programmet indeholder en del løse formuleringer (eksempelvis "ciffer i delnummer matcher ciffer i telefonnummer") som nødvendigvis skal gøres mere præcise i et rigtigt program. Husk på at formålet med programmet er at få computeren til



Figur 2: *Opdeling af problemet P i delproblemer.*

at problemet på en helt bestemt måde. Dette kræver stor præcision, og dermed et sprog hvori programmøren kan kontrollere den mindste detalje.

2.3 Opdel i flere delproblemer

Det første delproblem, hvis løsning vi netop har skitseret, er vanskeligt at løse. Det er fordi problemet faktisk er lidt for stort til at vi bør løse i et ubrudt forsøg. Når vi som programmører er i denne situation skal vi altid spørge os selv, om problemet endnu en gang kan deles op i to eller flere delproblemer. Det er næsten altid muligt! I vores problem kan det første delproblem f.eks. opdeles på denne måde:

1a. Er delnummeret placeret i starten af telefonnummeret?

1b. Er delnummeret placeret et sted i det telefonnummer der opstår ved at fjerne det første ciffer?

Det første delproblem er forholdvis let at løse. (Det svarer til arbejdet der laves i den inderste while-løkke i koden vist herover). Det andet delproblem ligner det oprindelige delproblem ganske meget - men det er blot mindre. Vi vender tilbage til denne iagttagelse senere i noten.

2.4 Forfin programmet trinvis

Når man står overfor en programmeringsopgave kan man spørge sig selv, hvordan løsningen bedst gribes an. Starter man blot “i øverste venstre hjørne”, arbejder

sig igennem alle detaljerne, og slutter med et punktum i "nederste højre hjørne"? Svaret er et klart *nej*. Som vi har set opdeler vi et stort problem (P) i et antal mindre problemer (så som P1 og P2) som så igen kan opdeles i endnu mindre problemer. I vores tilfælde blev P1 opdelt i P1a og P1b. Dette er illustreret i figur 2. Det anbefales ofte at starte med det øverste problem (problemet i toppen af figuren, P), og slutte med problemerne i bunden. En konkrete sådan fremgangsmåde kaldes for *top-down programmering ved trinvis forfinelse*.

2.5 Variable og imperativ programmering

I det program vi har skitseret vil der være nogle såkaldte *variable*, som "hele tiden" ændrer værdi. Det er eksempelvis de "tællere" som holder styr på hvilke cifre vi sammenligner i telefonnummeret og delnummeret. Et typisk programskridt ændrer værdien af en af disse variable.

Med et fint ord kaldes den form for programmering, vi har beskrevet, *imperativ programmering*. Ordet "imperativ" betyder "bydende" eller "kommanderende". Typiske kommandoer i imperativ programmering ændrer værdier af variable, eller skriver noget ud på skærmen. Der findes andre former for programmering som ikke fungerer gennem brug af kommandoer. Vi vil se et eksempel på dette i afsnit 6 om funktionsprogrammering.

Imperativ programmering: Programmering med kommandoer, hvor kommandoer kan ændre værdier af variable.

Variabel: Variable er pladser i programmets lager, som kan indeholde en bestemt værdi. I imperativ programmering kan en eksisterende variabel *tildeles* en ny værdi så ofte som det ønskes (f.eks. i en løkke).

Sekvens: Rækkefølgen af kommandoer er vigtig. Ofte vil det ændre programmets betydning hvis to kommandoer byttes om.

Gentagelse: Gentagelser programmeres med løkker, eksempelvis while-løkker. Ordet *iteration* bruges ofte i stedet for gentagelse.

Forgrening: Forgreninger programmeres med if-then-else. Forgreninger er udtryk for at programmet kan gå i to (eller flere) retninger. Ordet *selektion* bruges ofte i stedet for forgrening.

3 Programstruktur og abstraktion

Det vigtige budskab fra forrige afsnit var ideen om at opdele et problem i mindre delproblemer. Dette giver struktur i den måde vi løser problemet på - og det afleder en struktur i programmet. Der er næsten ingen programmer der skrives “i en lang smørre”.

3.1 Funktioner

Alle programmeringssprog indeholder faciliteter til programstrukturering. Den mest centrale - og mest basale facilitet - kaldes som regel for en *funktion*.[2] En funktion navngiver en programstump, eksempelvis hver af de to programdele som blev vist i afsnit 2. Ud over navnet styrer en funktion input til, og output fra funktionen.

Ideelt set skal alle de input data, som funktionen har brug for at arbejde på, overføres som *parametre*. Helt tilsvarende, skal alle funktionens resultater også sendes tilbage fra funktionen på en eller anden måde (gennem output parametre, eller gennem en særlig returværdi der knytter sig til funktionen). På denne måde bliver det muligt at have en ren grænseflade mellem en funktion og de omkringliggende programdele.

Lad os her skitsere en funktion som løser det første delproblem fra afsnit 2, uden dog at gentage detaljerne som er er gemt bag de tre prikker:

```
1  FUNCTION FindNummer(telefonnummer, delnummer): integer
2  BEGIN
3    ...
4  END
```

I linje 1 er telefonnummer og delnummer parametre. Input til funktionen FindNummer overføres gennem disse parametre.

[2] Funktioner kaldes i nogle sprog for *procedurer*, eller for *subprogrammer*. *Funktion* er faktisk en dårlig betegnelse, fordi det kan forvirre os i forhold til ægte og rene matematiske funktioner. Læs mere om dette i afsnit 6.

3.2 Kald af funktioner

I stedet for at skrive de to indlejrede while løkker, og alle detaljerne omkring dem, kan vi nøjes med at *kalde funktionen*. Ordet *integer* i slutningen af linje 1 fortæller at funktionskaldet vil give et heltal tilbage. Dette tal angiver placeringen af delnummeret i telefonnummeret. Med lidt "frækhed" kan man også, via det returnerede tal, formidle det mulige resultat at delnummeret ikke findes i telefonnummeret. I mit program anvender jeg et negativt tal til dette formål. Her er eksempel på et kald af funktionen FindNummer, som forventes at returnere tallet 4.

```
1  FindNummer(99408896, 08);
```

Hvis vi i vores program har behov for at finde mange delnumre af foreskellige telefonnumre er det smart blot at kunne kalde FindNummer, i stedet for at skulle gentage alle de detaljer som vi har set i afsnit 2.

3.3 Abstraktion

Alle programdetaljer skal placeres i stedet for de tre prikker i funktionen FindNummer. Vi siger at programdetaljerne *indskapsles* i funktionen. Vi vil også sige at funktionen *abstraherer* de programdetaljer der skal til for at finde delnummeret i telefonnummeret. Ved kald af funktionen FindNummer med passende parametre, og ved passende modtagelse af resultatet fra kaldet, kan vi nu se bort fra (glemme alt om) de detaljer, der skal til for at løse problemet. Det er nok for os at (1) kende navnet på funktionen, (2) vide hvilke parametre der skal sendes med til funtionen, og (3) vide hvordan resultatet kommer tilbage fra funktionen. Det er rigtig smart!

Alle løsninger på delproblemer, som er diskuteret i afsnit 2, indkapsles nu i hver sin funktion. Når vi bruger ideen om abstraktion kan vi gradvist glemme mange af de detaljer, som indkapsles i funktionerne. Det eneste vi har brug for er viden funktionernes navne og deres input/output grænseflader.

Der er ingen mennesker der kan rumme alle de detaljer der optræder i et stort program. I et stort program er det derfor af helt central betydning at vi bryder programmet op i funktioner, som abstraherer detaljerne. Med brug af ideen om abstraktion kan vi tillade os at se bort fra mange af disse detaljer - i det mindste så længe at alt går efter planen. Det er desværre ikke altid tilfældet. Og det er emnet for det næste afsnit.

Funktion: En funktion navngiver en programstump, således at kommandoerne i programstumpen kan køres ved blot at *kalde funktionen*.

Parameter: En parameter er en særlig variabel, der modtager input når en funktion kaldes.

4 Fejl og programtest

Fejl lurer overalt når vi skriver et program. Nogle fejl bliver aldrig fundet af programmøren, og sådanne fejl sniger sig derfor videre til brugere af programmet. Ofte til stor irritation og frustration.

Nogle fejl er åbenlyse, mens andre kan diskuteres. Hvis vores program f.eks. viser telefonnummeret 12345678 når vi beder om et nummer der indeholder cifret 9 er det en oplagt fejl. Men hvis vi får oplyst nummeret 92345679 når vi beder om et nummer der indeholder 99 kan det være udtryk for en fejl (fordi cifret 9 ikke optræder to gange i træk). Men det kan også være korrekt, fordi programmøren har en lidt anderledes forståelse af problemet (at de to 9 taller ikke nødvendigvis skal optræde ved siden af hinanden). Det er noget rod!

En god programmør skal være i stand til at anlysere et problem så grundigt, at han eller hun er helt på det rene med *hvad der er korrekt* og *hvad der er forkert*. Et problem kan kun løses i et program hvis vi har en nøjagtig og detaljere forståelse af problemet. Hvis problemet er stort og kompliceret er det en udfordring i sig selv at nå til dette punkt. Der kan således være mange forberedelser (*analyse*, opsamling og forståelse af *krav*, og *design*) inden selve programmeringen kan starte.

Hvis der er fejl i programmet skal programmet naturligvis rettes - jo før jo bedre. I forhold til programmering er det værdifuldt at fejl opdages så tidligt som muligt. Hvis fejl sniger sig ind i programmet, fordi vi ikke har helt styr på det problem vi skal løse, kan det blive meget tidkrævende (og dyrt) at få programmet til at virke korrekt. Oven i denne slags "forståelsesfejl" kommer andre former for fejl, måske fordi programmeren "ryster lidt på programmeringshånden". For at rette disse fejl skal vi "åben abstraktionerne", og vi skal igen forstå alle de detaljer som var nødvendige for at skrive programmet. Hvis det er lang tid siden vi skrev programmet, er dette ofte lige så vanskeligt - og lige så tidkrævende - som at skrive den første udgave af programmet.

En bestemt slags fejl kan helt forhindres i nogle programmeringssprog - de såkaldte

typefejl. Forestil dig at du kalder FindNummer med et personnummer i stedet for et telefonnummer, eller at vi forsøger at finde et fornavn i stedet for et delnummer. I et rigtigt programmeringssprog vil det fremgå at første parameter af FindNummer *skal* være et telefonnummer, og at sidste parameter *skal* være en del af et telefonnummer. Den slags fejl opdages automatisk i de fleste sprog, og dermed forhindres det at vi nogensinde kan køre et program som indeholder typefejl. Det er meget attraktivt!

Computeren ved altså, at du ikke kan kalde FindNummer på et personnummer, og at du ikke kan forsøge at finde et navn i stedet for et delnummer. Derfor er følgende kald af FindNummer *udelukket på forhånd*:

```
1  FindNummer(010212-1234, 123)
2  FindNummer(99408896, "Kaj")
```

Men hvordan finder du ud af om følgende kald, og deres angivne resultater, er korrekte?

```
1  FindNummer(99408896, 99) = 1
2  FindNummer(99408896, 96) = 7
3  FindNummer(99408896, 98) = -1
```

Løsningen er *teste programmet*, ved at prøvekøre det på de forskellige numre. Programmøren angiver - meget gerne inden selve programmet skrives - at ovenstående er *eksempler på korrekte resultater*. Når programmet er færdigt, testkøres det for at finde ud af om programmet opfører sig som forventet. Selve testkørslen kan automatiseres, så vi er fri for med håndkraft at kalde FindNummer og sammenligne det faktiske resultat med det forventede resultat.

Programtest er en meget vigtig aktivitet. I store programmer bruges der ofte mere tid på test end på programmering. På trods af dette, er en mængde af f.eks. 100 test kun en meget lille stikprøve. Forestil dig hvor mange test der skal udføres for at afprøve alle mulige delnumre på alle mulige telefonnumre. I vores program vil det tage timer at teste alle muligheder - selv på den hurtigste computer i verden. I de fleste andre programmer er der så mange forskellige muligheder, at det er umuligt at prøvekøre dem alle.

> **Opgave**: Tænk på en fejl du har oplevet på din computer. Giv et bud på om fejlen burde have været forhindret. Overvej om det vil være realistisk at finde fejlen ved en test.

5 Objekt-orienteret programmering

Programmer kan skrives på mange forskellige måder. Inden vi går over til at diskutere den objekt-orienterede måde vil vi karakterisere hvordan vi løste telefonnummer problemet imperativt i afsnit 2.

- Programmet blev skrevet med variable der ændrer værdi i takt med at programmets dele gentages i løkker.
- Programmet blev udviklet fra top til bund (top-down programmering ved trinvis forfinelse).
- Programmet blev struktureret i funktioner, der indkapsler programdetaljer, og som hæver abstraktionsniveauet.

Lad os nu se på objekt-orienteret programmering, der over de sidste par årtier har udviklet sig til at være den fremherskende måde at programmere på i IT-industrien. Udgangspunktet i et objekt-orienteret program, der løser telefonnummer problemet, vil være objektet Telefonnummer.

5.1 Objekter og operationer

Vi starter med at programmere objektet TelefonNummer. Det vigtigste i vores programmering af TelefonNummer er hvilke operationer vi kan udføre på et telefonnummer. Vi vil ikke i dette afsnit vise detaljerne i de beregninger, som gemmer sig bag operationerne. Her er nogle mulige operationer på telefonnumre, indrammet af selve objektet:

```
1 OBJECT TelefonNummer
2   OmraadeNummer(): integer
3   LokalNummer(): integer
4   FastnetNummer(): boolean
5   MobilNummer(): boolean
6   Ciffer(i): integer
7   RingTil()
8   SendSMS(besked)
9 END
```

De to første operationer udtrækker og returnerer dele af telefonnummeret, svarende til lokalnummeret og områdenummeret (i det omfang det giver mening). De to

næste fortæller om et telefonnummer er et fastnetnummer eller et lokalnummer.[3] Ciffer operationen udtrækker ciffer nummer *i* fra telefonnummeret, og returnerer dette som et heltal. Operationerne RingTil og SendSMS udfører kommandoer: Etablering af en forbindelse til telefonnummeret, som start på en telefonsamtale til dette nummer, og afsendelse af en SMS besked.

Tidligere har vi set på kald af funktioner med parametre. Hvordan kalder vi operationerne på et telefonnummer? Og hvordan kommer konkrete telefonnumre ind i billedet? Hvis vi antager at 99408896 repræserer et konkret telefonnummer objekt, kan vi udføre følgende operationer på det:

```
1  99408896.LokalNummmer()
2  99408896.MobilNummer()
3  99408896.Ciffer(8)
4  99408896.SendSMS("Det er sjovt at programmere")
5  99408896.RingTil()
```

Vi ser altså at det objekt, hvorpå der udføres en operation, står først, efterfulgt af en "dot" og navnet på den ønskede operation. Vi siger ofte at vi *sender en besked* til objektet. Første besked returnerer lokalnummer 8896, anden besked returnerer *false*, tredie besked returnerer 6 (som er det 8. ciffer). Beskeden SendSMS sender en kort tekst meddelelse til 99408896 under antagelse at nummeret er et mobilnummer. SendSMS returnerer ikke noget resultat som så. RingTil beskeden er en kommando, som heller ikke giver noget egentligt resultat tilbage. Der udføres derimod nogle komplicerede ting "bag scenen", der leder frem til at der kan startes en samtale til nummeret.

5.2 Datatyper og objekter

Vi har omtalt Telefonnummer som et objekt. Med dette mener vi ikke et bestemt telefonnummer, men telefonnumre i almindelighed. Det kan let forvirre! Derfor indfører vi *datatyper*. Datatypen telefonnummer betegner telefonnumre i almindelighed. Og fremover vil et telefonnummer *objekt* betegne et bestemt telefonnummer.

Objekt-orienteret programmeringen tager udgangspunkt i typen af data. Løsrevet fra et bestemt problem programmeres en mængde af alsidige operationer i den datatype, der er tale om. Side om side med andre datatyper (f.eks. Dato, Tidspunkt

[3] Et ja/nej svar håndteres i de fleste programmeringssprog ved typen boolean, der indeholder værdierne *true* og *false*.

og Kalender) vil typen TelefonNummer være i et bibliotek, som vi kan anvende i vores programmering. Tanken er, at når vi fremover har behov for at gøre et eller andet på et telefonnummer, vil den eksisterende datatype TelefonNummer umiddelbart løse vores behov. I en “rigtig fed udgave” af TelefonNummer typen kunne der også være en FindNummer operation, svarende til den funktion vi lavede tidligere i noten. I en “mindre fed udgave” vil vi skulle tilgå de enkelte cifre ved anvendelse af Ciffer operationen, og programmere en funktion FindNummer.

Når vi laver et objekt-orienteret program starter vi med at programmere de forskellige datatyper, vi har behov for i vores program. Datatyper indkapsler data på samme måde som funktioner indkapsler handlinger eller udtryk. Datatyper er abstraktioner, på samme måder funktioner. Datatyper indeholder funktioner, og som sådan hjælper datatyper med til at strukturere et program på et højere niveau end blot ved brug af funktioner.

Når vi har programmeret datatyperne, programmerer vi “en hat oven på typerne”, som udgør et specifikt program der løser vores faktiske problem. Et objekt-orienteret program udvikles således fra bunden mod toppen. Vi taler om “bottom up udvikling”, som kontrast til den “top down udvikling” vi beskæftigede os med i starten af noten i forbindelse med imperativ programmering.

5.3 Information hiding

En anden vigtig egenskab ved objekt-orienteret programmering er at dataegenskaberne af et telefonnummer er skubbet i baggrunden. Dette går ofte under betegnelsen *information hiding*. På et eller andet niveau i vores program skal vi finde ud af hvilke databestanddele der skal til for at arbejde med et telefonnummer. Langt inde i maven af datatypen TelefonNummer kan vi naturligvis finde disse detaljer. Men vi ønsker ikke at andre dele af vores program har viden om disse data detaljer.

Hvorfor nu dette hemmelighedskræmmeri? Fordi vi således på et senere tidspunkt kan ændre vores beslutning om data detaljerne, uden at påvirke andre dele af programmet end selve datatypen TelefonNummer. I meget store programmer er dette en stor fordel. Det svarer faktisk til at der i meget store bygninger er brandmure, som forhindrer at hele bygningen brænder ned, hvis vi skulle være så uheldig at tabe en tændstik.[4]

Den digitale hardware teknologi - selve elektronikken i computeren - er inspiratio-

[4] “At tabe en tændstik” i et program svarer til at “ændre mening om datarepræsentation”. Og tro mig - før eller senre taber vi en tændstik...

nen til imperativ programmering. Imperativ programmering “kører på samme melodi” som maskinkode programmering - blot på et lidt højere niveau. Hvad er inspirationen til objekt-orienteret programmering? Svaret er *begrebsdannelse* - altså måden hvorpå begreber beskrives og struktureres. Som sådan er objekt-orienteret programmering mere inspireret af en filosofisk tankegang end af en teknisk tankegang. Dette er da ganske tankevækkende i en verden, som i stor udstrækning er drevet af teknologiske ideer.

5.4 Specialisering

Har man forstand på begrebsdannelse vil man vide at det giver mening at tale om *specialisering* af et begreb. Eksempelvis er begreberne “kontorstol” og “lænestol” begge specialiseringer af begrebet “stol”. På samme måde som TelefonNummer og PersonNummer begge er specialiseringer af Nummer. I vores programmering vil nogle operationer i datatypen TelefonNummer stamme fra typen Nummer, og således også være tilgængelig på PersonNummer. Operationen Ciffer vil give mening på både personnumre og telefonnumre, og derfor vil vi placere Ciffer operationen i typen Nummer. Hvis vi har behov for meget specialiserede operationer på TelefonNummer (eksempelvis operationen FindNummer) vil vi selv kunne lave vores egen specialisering af klassen TelefonNummer.

Datatype: De fælles egenskaber ved en mængde objekter, eksempelvis telefonnumre. De vigtigste egenskaber er operationer.

Besked: En besked kalder en operation i et bestemt objekt.

Information hiding betegner ideen om at indkapsle detaljer, som kun kan ses inden i selve datatypen.

Specialisering: En specialisering af en datatype er en ny datatype, som arver generelle egenskaber, og som kan tilføje nye og mere specielle egenskaber.

6 Funktionsprogrammering

Fra matematik ved vi at en funktion er forskrift der på en entydig måde beskriver sammenhængen mellem et eller flere input (argumenter eller parametre) og ét output. En funktion afbilder altså et antal input værdier til netop én out-

put værdi. Sådan er det f.eks. med et polynomier, samt sinus og logaritmefunktionerne. Ud over afbildningsarbejdet har den matematiske funktion ikke andre påvirkningsmuligheder. Derved er den matematiske funktion anderledes end de funktioner, vi studerende i afsnit 2. Funktioner i imperative programmeringssprog vil ofte kunne påvirke variable på en sådan måde at disse variables værdiers ændres som følge af et kald af funktionen. Vi siger at sådanne funktioner har *sideeffekter*.

Spørgsmålet i dette afsnit er om vi kan skrive programmer baseret på egenskaberne ved matematiske funktioner - eller *rene funktioner* som vi ofte vil kalde dem. Svaret er "ja" - den kan vi sagtens gøre. Det er muligt at løses en meget bred vifte af problemer ved udelukkende at benytte rene funktioner.

Inden vi giver et eksempel vil vi berøre et par afgørende forskelle mellem imperativ programmering og funktionsprogrammering. I et funktionsorienteret program kan vi *binde et navn* til en værdi, men når først bindingen er foretaget kan vi ikke ændre bindingen efterfølgende. Navnet bliver altså ved med at have den værdi, den oprindeligt er bundet til. I funktionsprogrammering har vi altså ikke variable, som kan tildeles nye værdier i takt med at programmet kører.

6.1 Løkker eller rekursion?

En anden væsentlig forskel er måden hvorpå vi håndterer gentagelser og løkker, som jo var helt centrale i den programmering vi stiftede bekendtskab med i afsnit 2. I funktionsprogrammering laver vi gentagelser ved at funktionen kalder sig selv. Ved først øjekast kan dette virke meget "syret", vanskeligt at forstå, og noget anderledes end det vi genkender fra gængse matematiske funktioner. Men om to sekunder vil vi se, at dette for det meste er helt naturligt i forhold til de problemer vi løser.

Lad os vende tilbage til telefonnummer problemet fra afsnit 2. Delproblem nummer 1 er finde ud af hvor et delnummer er placeret i telefonnummeret (og om det overhovedet findes i telefonnummeret). Vi indså at dette delproblem med fordel kan deles op i to delproblemer, nemlig følgende:

1a. Er delnummeret placeret i starten af telefonnummeret?

1b. Er delnummeret placeret et sted i det telefonnummer der opstår ved at fjerne (eller se bort fra) det første ciffer?

Som allerede observeret er delproblem 1a forholdsvis enkelt at løse. Det bemær-

kelsesværdige er at delproblem 1b svarer fuldstændigt til (det oprindelige) problem 1 - det er blot lidt mindre, fordi vi arbejder med et telefonnummer der har ét ciffer mindre end det tidligere telefonnummer. På et tidspunkt, når vi har "høvlet" tilstrækkeligt mange cifre af, er der ikke flere cifre tilbage. På dette tidspunkt løser problemet næsten sig selv.

Den funktion som løser delproblem 1 kan således anvendes til at løse delproblem 1b. Den pågældende funktion kan således kalde sig selv. Vi siger at funktionen er *rekursiv*. Vi har med et problem at gøre, hvor *problemet i sin helhed* også optræder som et *delproblem i detaljen*!

Generelt er det således i funktionsprogrammering, at rekursive funktioner tilfredsstiller vores behov for gentagelser. Der er ingen while-løkker i rene funktionsprogrammer.

6.2 Et eksempel på en rekursiv funktion

Vil vil nu skitsere funktionen der løser delproblem 1b. Den hedder stadig FindNummer, og den arbejder på et telefonnummer og et delnummer ligesom i afsnit 3.

```
1  FUNCTION FindNummer(telefonnummer, delnummer): integer
2  BEGIN
3     IF (TomtTelefonNummer(telefonnummer))
4     THEN −20
5     ELSE IF (StarterTelefonNummerMed(telefonnummer, delnummer))
6     THEN 1
7     ELSE 1 + FindNummer(AfkortNummer(telefonnummer), delnummer)
8  END
```

Da vi gradvist fjerner cifre fra forenden af telefonnummeret bliver vi i linje 3 og 4 nødt til at tage højde for situationen hvor telefonnummeret er tomt. Dette er meget karakteristik for programmering med rene, rekursive funktioner. Vi returnerer i så fald et passende negativt nummer (se opgaven herunder). Hvis telefonnummer starter med delnummmer er resultatet selvfølgelig 1, som det fremgår i linje 5-6. Hvis ikke telefonnummer starter med delnummeret (og hvis det ikke er tomt) kalder vi FindNummer rekursivt på et telefonnummer, hvor det første ciffer er fjernet (som udført af den rene funktion AfkortNummer). Dette ses i linje 7. Dette kald forsøger at finde delnummeret i det nu afkortede telefonnummer. For at kompensere for at vi nu finder delnummeret i det afkortede nummer, lægger vi 1 til resultatet af det rekursive kald.

Hjælpefunktionerne TomtTelefonNummer, StarterTelefonNummerMed og AfkortNummer er alle småfunktioner. Da de løser ganske små delproblemer er de lette at have med at gøre. Vi vil ikke her gå nærmere i detaljer med dem.

> **Opgave**: Det ser mystisk ud at vi returnerer -20 hvis telefonnummeret er tomt. Og der er faktisk ikke særligt pænt, som det er skrevet ovenfor. Det er lidt af et trick! Kan du indse "hvor lille" det negativt nummer skal være relativt til antal af cifre i telefonnummeret for at FindNummer giver os et negativt resultat, hvis ikke delnummer findes i telefonnummer?

> **Opgave**: Forestil dig at kaldet FindNummer(99408896,88). FindNummer kalder FindNummer, som igen kalder FindNummer, som igen...
>
> Forsøg på at forestille dig hvordan beregningen forløber, og gør rede for hvordan resultatet 5 fremkommer.
>
> Følg tilsvarende detaljerne i kaldet FindNummer(99408896,77). Hvilket tal kommer tilbage fra dette kald?

Det andet delproblem består, som du nok husker, i at finde de numre i telefonbogen, som indeholder delnummeret. Dette problem kan løses ved at anvende funktionen FindNummer på hele telefonbogen. Vi siger ofte at vi *mapper* funktionen FindNummer på listen af telefonnumre i telefonbogen. Det viser sig at mapningen i sig selv er udtryk for en gentagelse, som kan programmeres ved en rekursiv funktion. Tænk lige over det!

Ren funktion: En funktion som ikke har side-effekter, og som ikke ændrer på værdier af variable.

Rekursiv funktion: En funktion, der kalder sig selv. Rekursionen er udtryk for at det problem P, som funktionen løser, optræder som et delproblem af P.

7 Programmeringsarbejdet

Det kan være svært for "almindelige mennesker" at forestille sig hvordan en programmør - eller systemudvikler - arbejder. Lad os her, som afslutning af noten, diskutere forskellige sammenligninger, for at komme dette lidt nærmere.

Kan vi sammenligne programmering med matematik - ligner tankegangen af en programmør din matematiklærers måde at tænke på? Logisk og systematisk tankegang er vigtige egenskaber af en god programmør. Og som sådan er der et overlap mellem den gode matematiker og den gode programmør. Nogle former for programmering er direkte sammenlignelige med at bevise en matematisk sætning. Men denne tankegang er dog ikke fremherskende i praktisk programmering. Sammenfattende vil jeg sige, at nogle matematiske dyder fremmer god programmering; Men programmering bør ikke ligestilles med matematik. Der er mange gode programmører som er elendige matematikere.

Programmer bliver oversat fra et programmeringssprog til maskinsprog, som umiddelbart kan udføres på en computer. Vi kan derfor spørge om en programmør derfor arbejder på samme måde som en tolk? Svaret er et rungende "nej". Oversættelsesarbejdet fra et højniveau programmeringsssprog til maskinekode er automatiseret. Der er altså skrevet et program - en oversætter eller compiler - der tager sig af dette arbejde. I mange moderne udviklingsmiljøer skal programmøren slet ikke tænke over oversættelsesarbejdet.

Kan vi sammenligne en programmør med en håndværker, som er faglært til at arbejde med f.eks. mursten, jern eller træ? På nogle måde er denne sammenligning nyttig og frugtbar. En klassisk håndværker (såsom en murer, smed eller tømrer) forarbejder et materiale med omhyggelig brug af forskellige former for værktøj. På samme måde har en programmør brug for værktøj for at konstruere og bearbejde software. For både håndværkere og programmører tager det tid at beherske håndelaget, så der kommer gode produkter ud af arbejdet. Bemærk i denne sammenhæng programmørens værktøjer er ikke fysiske værktøjer, men snarere programmerede værktøjer såsom editorer, compilere, test værktøjer, og ganske komplicerede integrerede udviklingsmiljøer.

Når der skal laves et nyt IT-system til f.eks. politiet sker det på baggrund af en lang kravspecifikation. Efter en licitation bygger systemudviklerne i et stort IT-firma politiets nye IT-system. Det koster rigtig mange penge - og trist nok lykkes byggearbejdet ikke altid. I denne form for systemudvikling og programmering minder arbejdet om designarbejde, ingeniørarbejde og arkitektarbejde.[5] Efter min opfattelse er denne analogi den er virker bedst. I de fleste sammenhænge svarer programmering til ingeniørarbejde. Det er ikke tilfældigt at den største mængde studerende, som uddannes i Institut for Datalogi på Aalborg Universitet, betegnes som *softwareingeniører*.

"Rigtige ingeniører" bygger broer, tårne eller IC4 tog. Materialet for disse in-

[5] Sammenlignet med håndværkeren er en ingeniør ofte både praktisk og teoretisk udddannet. Det teoretiske fundament er vigtigt for de fleste programmører.

geniører er håndgribeligt, hårdt og fast. Softwareingeniører bygger IT-systemer, hvor programmer er den "faste bestanddel". Men programmerne er i sig selv uhåndgribelige, og helt anderledes af natur end f.eks. Storebæltsbroen, Eiffeltårnet, eller et IC4 tog. Et program kan kopieres på et splitsekund; Det tager 10 år at kopiere storebæltsbroen; Og der findes programmer som har været lige så dyre og besværlige at bygge som Storebæltsbroen. Programmer er udfordrende og fascinerende!

Igennem mange år har ingeniørarbejdet været grundlaget for opbygningen af det moderne industrisamfund. I dag spiller softwareingeniørens arbejde en tilsvarende rolle for udbygningen af informationssamfundet. Fremover vil det kun gå én vej: Der bliver behov for endnu flere programmører, som kan konstruere endnu flere applikationer på vore talrige små og store computere.