Pseudokode.

Indledning.

Når man konstruerer et program er der mange aspekter man skal holde styr på: Hvad skal programmet egentlig gøre, interaktionen mellem bruger og program, skærmbilledernes udseende, datastrukturer, formater på filerne osv. Til at hjælpe med at håndtere dette er der i tidens løb lavet en mængde værktøjer. Man kan godt få det indtryk at enhver amerikansk programdesignguru med respekt for sig selv, har lavet sin egen metode og ivrigt sælger de tilhørende bøger og kurser.

Ingen af de mange værktøjer er blevet *værktøjet*, som alle er enige om at bruge. Det tjener derfor ikke noget formål i programmeringsundervisningen at bruge en bestemt guru's metode. Men på den anden side kan man ikke klare sig uden et værktøj til at hjælpe sig i konstruktionsprocessen. Derfor vil vi indføre et lille simpelt værktøj som kaldes *pseudokode*. (Det udtales "søudokode".)

Den følgende beskrivelse bygger på Thorkild Skjelborg :"Datalogi: Programmering i Pascal". Oprindelig skrev jeg disse noter til et hold som brugte PASCAL; men det er en vigtig pointe at metoden er uafhængig af programmeringssproget!

Hvornår skal man bruge pseudokode?

Pseudokode bruges til at konstruere algoritmer til at løse et allerede fastlagt problem. Det er altså ikke et værktøj til at designe skærmbillede eller fastlægge bruger-program storyboard eller nogen af alle de andre aspekter af programkonstruktionen. Metoden bruges til at finde ud af hvordan man kan få sit program til at udføre de ønskede handlinger.

Metoden er ment som en hjælp til at konstruere kode til løsning af komplicerede problemer. Hvis problemet er så let, at man kan lave koden med det samme, skal man ikke bruge pseudokode.

Hvorfor kaldes det pseudokode?

Beskrivelsen af de handlinger, som skal til for at løse opgaven, skal være både generel og letforståelig. Blandt andet derfor bruger vi *almindelige danske ord* til beskrivelsen. På den måde undgår man også at spilde tid på at overveje særlige detaljer i programmeringssproget, som f.eks. om der skal bruges komma eller semikolon mellem de variable i en procedureerklæring.

Samtidig med at man bruger danske ord til beskrivelsen af delopgaverne, skriver man disse ord i en opsætning, som minder om programmeringssprog. Derfor navnet *pseudokode*. Teksten bliver en slags kode ligesom PASCAL. Der findes blot ingen programmer, som kan oversætte det til maskinkode.

Ved at vælge en beskrivelse som ligger midt imellem daglig tale og f.eks. PASCAL eller Basic, får man slået bro mellem på den ene side det sprog, opgaven er beskrevet i, og på den anden side det sprog, som løsningen skal skrives i.

Princippet bag pseudokode.

Formålet med pseudokode er som sagt at finde en algoritme som løser en fastlagt opgave. Princippet i pseudokode er det gamle "Del og hersk.". Hvis opgaven er simpel nok, skriver man blot noget kode, som klarer sagen. Hvis opgaven er kompleks, opdeler man den i små delopgaver, som hver især er så simple, at man kan skrive kode til dem.

Dette lyder jo nemt nok, men hvis du tænker dig om, vil du indse, at der nu er to nye problemer: Dels hvordan skal opdelingen i delopgaver laves og dels hvordan undgår man at miste overblikket med alle disse delopgaver?

Problemet med hvordan man opsplitter sin opgave i mange små delopgaver, klarer man ved at anvende metoden på metoden selv: man laver denne opsplitning lidt ad gangen i overskuelige trin.

Problemet med at bevare overblikket klarer man ved at bruge grafik. Man indrammer simpelthen de dele som hører sammen. Mennesker er nemlig bedst til at skaffe sig overblik ved at se på tegninger og diagrammer.

Selve metoden.

For at holde styr på hvor langt man er i denne opsplitning af problemet, beskriver man løsningen i en række udgaver, hvor den første er meget generel, og de efterfølgende er stadig mere detaljerede.

I sin første udgave af løsningen deler man opgaven op i nogle få (typisk 3 til 7) delhandlinger. Det er vigtigt at tænke i *handlinger*. I sidste ende ønsker man jo at edb-maskinen skal *gøre* noget.

De delhandlinger, som er komplicerede, splitter man yderligere op i 2.udgave, og sådan fortsætter man. Til sidst er alle delopgaverne så simple, at man umiddelbart kan skrive den nødvendige kode i det valgte programmeringssprog.

Efterhånden får man fastlagt navne og typer på de variable. Hvis datastrukturen er kompliceret må man supplere med tegninger. Klumper af handlinger, som udgør en logisk helhed, identificeres og udskilles som separate navngivne procedurer eller funktioner. Hver af disse procedurer kan så konstrueres færdig separat.

Processen stopper når man føler, at alle dele af pseudokoden er klar til at blive skrevet i det valgte programmeringssprog.

Skrivemåde

En vigtig del af pseudokodemetoden er at bruge en bestemt opstilling for at danne sig et grafisk overblik.

En **sekvens** af handlinger som skal udføres efter hinanden skrives under hinanden i rækkefølge.

```
…..
…..
```

Forgreninger skrives som:

```
hvis ……
så
    ┌─────┐
    │ ….. │
    │ ….. │
    └─────┘
ellers
    ┌─────┐
    │ ….. │
    │ ….. │
    └─────┘
```

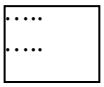
Her markerer rammerne de handlinger som skal udføres sammen.

Løkker kan skrives på mange måder her er fire eksempler

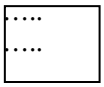
```
Gentag
    ┌─────┐
    │ ….. │
    │ ….. │
    └─────┘
indtil alle tal er brugt
```

```
Sålænge der er flere bogstaver
    ┌─────┐
    │ ….. │
    │ ….. │
    └─────┘
```

Gentag for hver linie i filen



For hvert ulige tal mellem 33 og 72



Ikke den første, men den bedste løsning.

Hver gang man har lavet en ny udgave skal man stoppe op og overveje, om det nu også er den bedste løsning man har fundet. Findes der alternativer som måske er bedre? Nogle alternativer kræver blot små ændringer, andre kræver at programmets grundlæggende struktur laves om.

Det er idealet, at man undervejs identificerer den bedste løsning på hvert detaljeringsniveau, så man ikke fortsætter med flere og flere detaljer i en løsning, som alligevel ikke var den rette. I virkelighedens verden opdager man somme tider først under aftestningen af programmet, at løsningen er forkert! Man skal så vende tilbage til sin pseudokode og se hvor den skal laves om. Men hvis man er omhyggelig i hvert trin kan man undgå den slags ærgerlige fejl.

Husk: ”Den ultimative form for dovenskab er at gøre tingene rigtigt første gang.”

Eksempel på brug af pseudokode.

Opgaven.

Opgaven jeg har valgt til eksemplet, er simpel, men rummer alligevel problemer nok til at I kan se, hvordan pseudokode bruges. Jeg tænker på en situation hvor opgavestilleren modtager post fra en afsender som ikke har et dansk tastetur til sin computer. Opgavestilleren oensker at slippe for at laese tekst som ser saadan her ud.

Opgaven kan præciseres til at læse en tekst fra en tekstfil og erstatte alle forekomster af ”ae” med ”æ”, af ”oe” med ”ø” og af ”aa” med ”å”. I første omgang ser vi bort fra problemerne med store bogstaver.

1. udgave.

åben tekstfilen
læs teksten og konverter til danske bogstaver
skriv i en ny fil
luk filen

Læg mærke til at dette faktisk er en løsning på opgaven, den er blot ikke detaljeret nok til at blive omskrevet til programmeringssprog endnu. Jeg kan allerede nu fastlægge lidt af datastrukturen: Der er en tekstfil med den oprindelige tekst, den vil jeg kalde `indfil` og den nye tekstfil `udfil`.
Linie nummer to er stadig for abstrakt, så den bliver udbygget i næste udgave.

2. udgave.

åben `indfil`
sålænge der er flere linier i `indfil`

> læs næste linie i `indfil`
> konverter linien
> skriv den konverterede linie til `udfil`

luk `indfil`

Løkken ”sålænge…” er et eksempel på hvordan grafikken bruges til at vise præcis hvilke linier som skal gentages.
Her kan man også straks se en fejl: Hvorfor nævnes `udfil` så lidt? Det er et eksempel på hvordan metoden sikrer at man bevarer overblikket og får rettet sine fejl. 1.udgave skulle have heddet: ”åben begge tekstfiler” osv.

Det er også tydeligt, at jeg får brug for et sted at have den linie, der er ved at blive konverteret, så jeg indfører en string variabel `linie`.

3.udgave.

åben `indfil` og opret `udfil`
sålænge der er flere linier i `indfil`

> læs næste linie i `indfil` og anbring den i `linie`
> konverter teksten i `linie`
> skriv `linie` til `udfil`

luk `indfil` og `udfil`

Det gav ikke meget nyt, så jeg må åbenbart nu gå i gang med den egentlig konvertering. Konverteringen er blevet isoleret til " konverter teksten i `linie`" så jeg vil nu arbejde videre på den alene. Muligvis vil det blive til en separat procedure i den endelige kode. Ind til videre vil jeg kalde den `konverterTekst.`

konverterTekst, udgave 1.

læs bogstaverne i `linie`
hvis det er ae, oe eller aa så erstat med æ,ø og å

Her er der problemer! For det første: Hvad vil det egentlig sige at læse bogstaverne i linien? Skal de læses ét ad gangen eller dem alle sammen på en gang? For det andet: Hvornår skal "hvis det er…" udføres, én gang eller efter hvert bogstav? For det tredje: "hvis det er…" hvad referer ordet "det" egentlig til?

Problemerne skyldes ikke at jeg ikke har detaljer nok, men at jeg har været alt for upræcis! så:

konverterTekst, udgave 1 rettet.

læs bogstaverne i `linie` et ad gangen og for hver bogstav

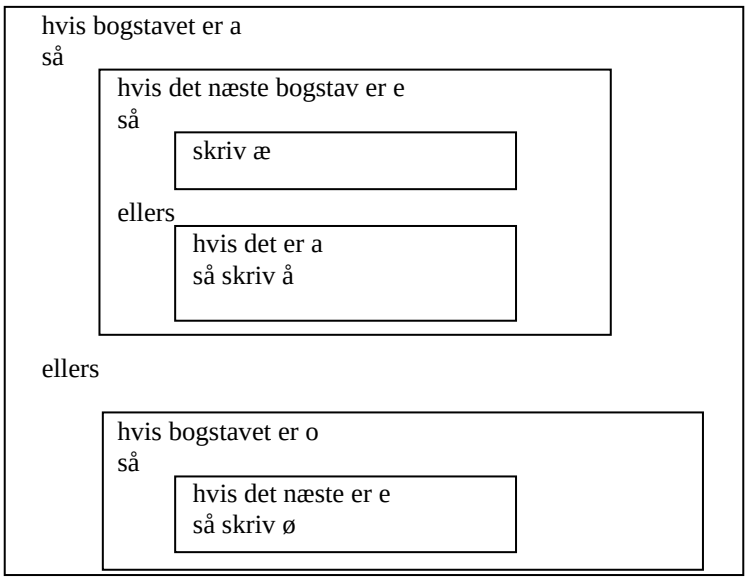
hvis det er ae, oe eller aa så erstat med æ,ø og å

Det er mere præcist, - så præcist at jeg klart kan se, at jeg har lavet en tanketorsk: Hvordan kan ét bogstav være f.eks. ae ?

Det er efter min mening en af pseudokode-metodens store fordele: Man kan fange sådanne designmæssige brist længe før, man har skrevet metervis af kode. Kode som man dels må kassere og dels mentalt er fanget af, så det kan være næsten umuligt at få øje på alternativerne.

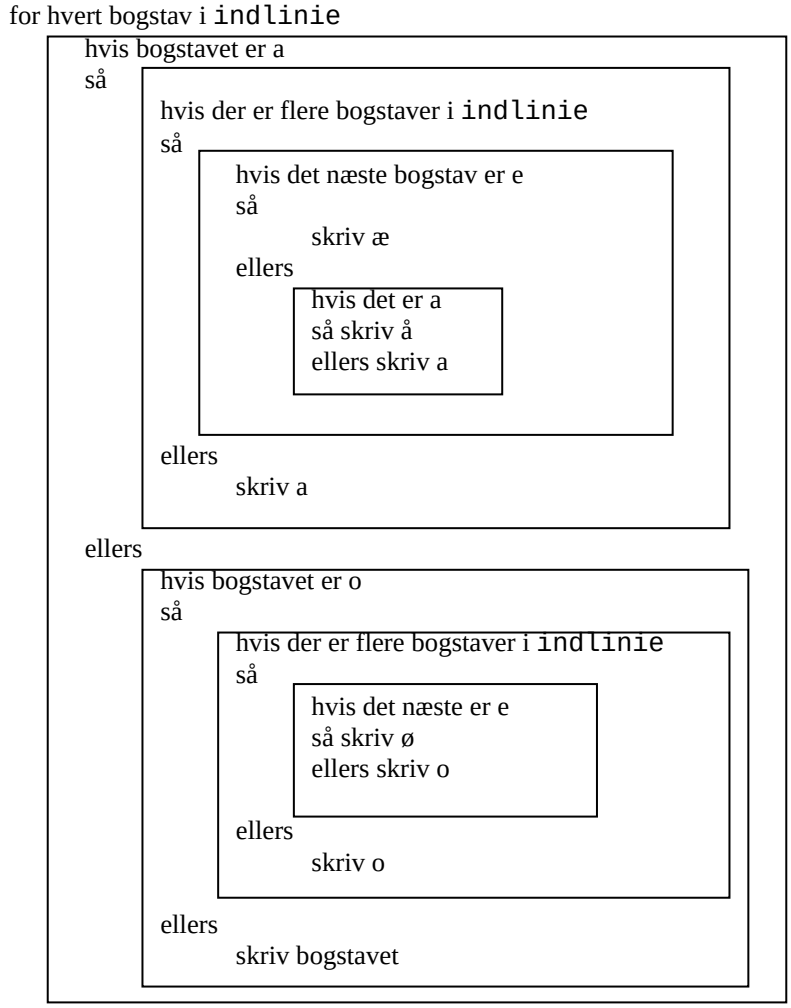
konverterTekst, udgave 2.

for hvert bogstav i `linie`



Her ses strukturen tydeligt med forgreninger inden i hinanden. Men det bliver værre for ”hvis det næste bogstav …” kræver jo at der *er* et næste bogstav. Der skal derfor undersøges om enden af linien er nået.
Ordene ”skriv æ” gemmer også problemer: Hvor skal æ skrives, i stedet for a? eller i stedet for både a og e? Hvis jeg gør det sidste så skal resten af teksten i `linie` rykkes en plads til venstre, og det vil igen kræve noget kode. Jeg valgte at løse dette ved at indføre to variable `indlinie` og `udlinie` i stedet for blot `linie`.

konverterTekst, udgave 3.



På dette sted i programkonstruktionen overvejede jeg om dette virkelig er den letteste og eleganteste løsning. For at afklare dette afprøvede jeg nogle alternativer:

konverterTekst, udgave 3 B.

for hvert bogstav i `linie` undtagen det første

```
hvis bogstavet er e
så
        hvis det foregående bogstav var a
        så
                skriv æ
…
…
```

Denne løsning kan straks forkastes da den vil give en fejl: Kaere bliver til Kaære. For at reparere på det skal jeg slette fra udlinie. Det kan lade sig gøre, men det bliver rodet. Og både ideen om at reparere på løsningen og det rodede indtryk er advarselslamper om, at her skal der overvejes alternativer.

Men ideen bag løsningen er god nok, så den bliver brugt i er nyt alternativ:

konverterTekst, udgave 3 C.

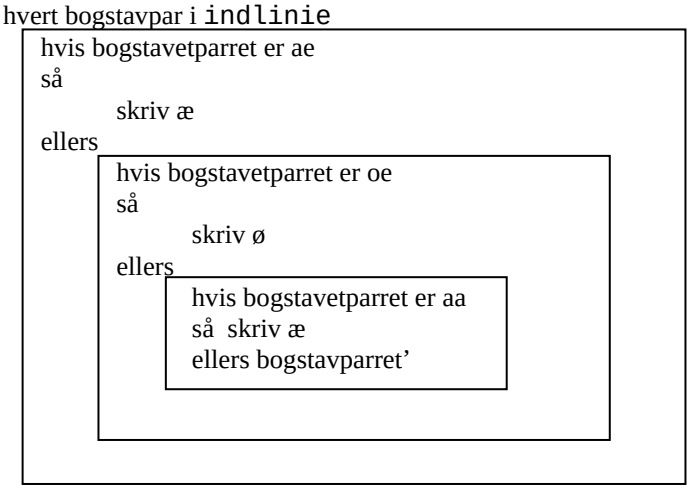
for hvert bogstav i `indlinie` undtagen det sidste

```
hvis bogstavet er a
så
        hvis det næste bogstav er e
        så
                skriv æ
        ellers
                hvis det næste bogstav er a
                så skriv å
                ellers skriv a
ellers
        hvis bogstavet er o
        så
                hvis det næste bogstav er e
                så  skriv ø
                ellers skriv o
        ellers
                skriv bogstavet
skriv det sidste bogstav i linien
```

Dette ser mere lovende ud! Men måske kan det forbedres.

konverterTekst, udgave 3 D.

for hvert bogstavpar i `indlinie`



skriv om nødvendigt det sidste bogstav i linien

Denne løsning er let læselig og den er logisk og klar. Men er det muligt at løbe bogstavpar i linien igennem? Det kræver selvfølgelig at man prøver at skrive den ned med flere detaljer.

Jeg stopper her. Formålet var jo slet ikke at lave et program; men at vise pseudokode metoden og den skulle gerne nu stå klart.

Opfordring fra undervisningsudvalget:

Hvis du bruger noten, så send en mail til Per Holck på pho@ats.dk, som har stillet noten frit til rådighed.