Kommunedata Brøndby-centralen

Jan Weimann

STRUKTURERET DESIGN

JULI 1975

Kommunedata Brøndby-centralen

STRUKTURERET DESIGN

JULI 1975

FORORD

Materialet til denne artikel er hentet fra IBM's system journal, VOL.13 No.2 1974.

Den er forfattet af W.P. Stevens, G.L. Meyers samt L.L. Constantine. Oversat til dansk ved undertegnede.

Der er i artiklen omtalt en del nye begreber. Navngivningen af disse står helt for egen regning; der er ikke tale om forsøg på at danne standards i KMD, det arbejdes der på andetsteds i firmaet.

Hans Lind
Kommunedata
Juli 1975

I N D H O L D S F O R T E G N E L S E

INDLEDNING

Struktureret design er fremlagt som et sæt af generelle konstruktionsprincipper og -teknikker for at gøre kodning, fejlretning og modificering lettere, hurtigere og billigere ved at mindske kompleksiteten.

Hovedideerne i dette, er et resultat af næsten 10 års forskning udført af L.L.Constantine.

Resultaterne af denne forskning er præsenteret her, men forfatterne har ikke i sinde at præsentere teorien og oprindelsen af resultaterne i dette skrift. Disse sider er blevet kaldt "Composite design" af G.J. Meyers. Forfatterne mener, at disse program design teknikker kan kombineres med, og bliver bedre med dokumentations-teknikkerne kaldet HIPO, og kodnings-teknikkerne kaldet struktureret kodning.

Disse omkostningsbesparende teknikker skal nødvendigvis altid anvendes i balance med andre hensyntagen og krav til systemet. Men muligheden for at udarbejde enkle programmer der er lette at ændre, bliver mere og mere aktuel, efterhånden som prisen på programmeringen stiger.

GENERELLE OVERVEJELSER VED STRUKTUREret DESIGN

Enkelhed er den mest anbefalelsesværdige "målestok" til brug ved udvælgelse af alternative design-muligheder set i relation til reduceret tid ved fejlfinding og ændring. Enkelheden kan forøges ved at dele systemet i separate dele på en sådan måde, at hver del kan betragtes, implementeres, udarbejdes og ændres med minimal indflydelse på andre dele af systemet. Observerbarhed (muligheden for let at opfatte hvordan og hvorfor tingene sker) er en anden nyttig betragning, der kan hjælpe til at designe programmer, der let kan ændres. Betragtninger i retning af effekten af rimelige ændringer er også værdifuld.

L.L. Constantine har observeret, at de programmer der var lettest at implementere og ændre, var dem der bestod af simple, uafhængige moduler. Grunden til dette er, at løsning af et problem er lettere, når problemet kan underopdeles i dele der kan betragtes separat. Problemløsning er sværest når alle aspekter af problemet skal betragtes samtidig.

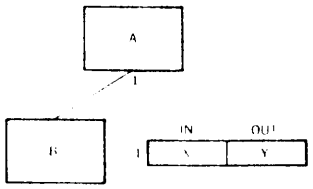
Betegnelsen modul bruges til at betegne et sæt af et eller flere sammenhængende program statements, som har et navn således at andre dele af systemet kan aktivere det, og som fortrinsvis har sine egne specifikke sæt af variabel navne. Exempler på moduler er PL/1 procedurer, FORTRAN, "mainlines and subprograms", og generelt, subrutiner af alle slags. Betragtninger er altid med relation til program statements som de er kodet, da det er programmørens mulighed for at forstå og ændre source-programmet, der er under overvejelse.

Overvejelser i retning af at dele programmer op i mindre dele er særdeles nyttige, men de teknikker der præsenteres her, er beregnet til brug ved design af simple, uafhængige moduler "fra fødslen". Det viser sig at være vanskeligt at dele et existerende program op i separate dele uden at forøge komplexiteten, på grund af mængden af kode der overlapper hinanden samt andre indbyrdes afhængighedsforhold.

Grafisk afbildning er et nyttigt værktøj ved struktureret design. Figur 1 beskriver en sådan, kaldet et "structure chart" (struktur diagram), hvor:

1. Der er to moduler, A og B.
2. Modul A aktiverer modul B. B er underordnet til A.
3. B modtager en input parameter x (dens navn i modul A) og returnerer en parameter Y (dens navn i modul A). (Det er vigtigt at bestemme hvilke parametre der repræsenterer data der overgives til det kaldte program, samt hvilke der er til data der returneres til det kaldende program).

Figur 1.



KOBLING OG KOMMUNIKATION

Når man skal vurdere alternativer ved opdeling af programmer i moduler, er det vigtigt at analysere og vurdere typer af "forbindelser" mellem moduler. En sådan forbindelse er en reference til en alle anden label eller adresse der er defineret et andet sted.

Jo færre og simplere forbindelserne er mellem modulerne, jo lettere er det at forstå hvert modul uden at referere til andre moduler.

Når man minimerer forbindelser mellem moduler, minimerer man samtidig mulighederne for at rettelser og fejl svulmer op til at forårsage fejl i andre dele af systemet. D.v.s. eliminerer katastrofale "onde cirkler", hvor ændringer i en del af systemet forårsager fejl i en anden, der nødvendiggør yderligere ændringer, der forårsager nye fejl o.s.v.

Den meget udbredte teknik, brug af "fælles" data arealer (eller globale variable, eller moduler uden eget specifikt sæt af variable navne), kan resultere i enorme mængder af forbindelser mellem moduler i et program. Kompleksiteten i et system er afhængig ikke alene af antallet af forbindelser, men også af hvor "stærkt" hver forbindelse sammenkobler (forener) to moduler, idet jo stærkere forbindelse, jo større afhængighedsforhold. Kobling er målestokken for hvor stærk foreningen, etableret ved forbindelser mellem to moduler, er. Stærk kobling komplicerer et system, fordi et modul i sig selv er sværere at forstå, ændre og rette, hvis det er meget afhængig af andre moduler. Kompleksitet kan reduceres ved at designe systemer med svagest mulig kobling mellem moduler.

Styrken af kobling etableret ved en given forbindelse er bestemt af flere faktorer, og det er således svært at etablere en simpel "gradinddeling" af kobling. Kobling afhænger af (1) hvor kompliceret forbindelsen er, (2) om forbindelsen refererer til modulet selv eller noget inden i modulet, og (3) hvad bliver sendt og hvad bliver modtaget.

Styrken af kobling stiger ved stigende kompleksitet, eller uklarhed omkring grænseflader (det der forbinder) mellem moduler. Kobling er svagere når forbindelsen er til modulets normale grænseflade (modulets normale "indgang") end når den er til et internt komponent i modulet. Kobling er svagere ved data forbindelser end ved kontrol forbindelser, hvilke igen er svagere end hybride (blandede, kompliceredе) forbindelser. Når to eller flere modulers grænseflade er det samme areal at storage, data region, eller enhed, siges de at dele et fælles "environment". Eksempler på fælles environments er:

- et sæt af data elementer med EXTERNAL attribute der INCLUDE's ind i PL/1 moduler eller findes listet i hvert af et antal moduler.
- data elementer defineret i COMMON statements i FORTRAN moduler.
- en centralt placeret "control block" eller sæt af "control blocks".
- en fælles overlay region af lageret.
- globale variabel navne defineret over et helt program eller en hel program sektion.

Den vigtigste strukturelle karakteristik ved et fælles environment er, at det kobler hvert modul der deler det til et hvert andet sådant modul, uden hensyn til deres funktionelle forhold i systemet. For eksempel kun de to moduler XVECTOR og VELOC gør rent faktisk brug af element X i et included fælles environment i PL/1, men ændring af længden af element X vedrører alle moduler der bruger elementer i dette fælles environment og må således re-kompileres.

Hvert element i det fælles environment, brugt af givne moduler eller ej, danner specifikke "stier" gennem systemet ad hvilke fejl og ændringer kan brede sig. Hvert element i det fælles environment forøger kompleksiteten i det totale system, og komplek-

siteten øges med hvert par af moduler der deler dette environment. Ændringer til og ny brug af det fælles data, vedrører moduler på nærmest uforudsigelig måde. Data referencer bliver ukontrollable, ja sommetider ukendte.

Et modul hvis grænseflader er et fælles environment for nogle af dets input eller output data,er generelt mere indviklet at bruge i forskellige sammenhæng end et modul, hvis kommunikation er begrænset til parameteroverførsel ved call's. Det er mere klodset at etablere en ny eentydig data sammenhæng (forbindelse) ved hvert kald af et modul, når data overførsel sker ved fælles environment.

Uden analyse af alle implicerede moduler eller omhyggelig "opbevaring" af værdier, bliver nyanvendelse af det fælles environment det samme som at blande sig i andre moduler, og fejlene vælter frem i systemet. Også med hensyn til fremtidige udvidelser i et system; når man een gang har valgt at bruge fælles environments, bliver alle nye moduler afhængige af det. Det forøger kompleksiteten yderligere. På dette punkt har Belady og Lehman observeret, at "et velstruktureret system i hvilket kommunikation foregår via parameteroverførsel gennem definerede grænseflader, oftest er lettere at udvide og kræver mindre indsats til vedligeholdelse".

Det er muligt at minimere ulemperne ved fælles environments. Det kan gøres ved at begrænse adgang til disse til det mindst mulige antal moduler. Hvis man deler samtlige fælles elementer op i grupper, der hver for sig er krævet af et antal moduler, så reducerer man såvel størrelsen af de fælles arealer, som antallet af moduler der kræver brug af dem. Brug af navngivne i stedet for blanke COMMON i FORTRAN er et eksempel på dette.

Kompleksiteten i forbindelse med modulers grænseflader er bestemt af, hvor meget information der er nødvendigt for at etablere og forstå forbindelser. Når indbyrdes relationer er klare og tydelige, bliver koblingen svagere end når disse relationer er uklare og indviklede.

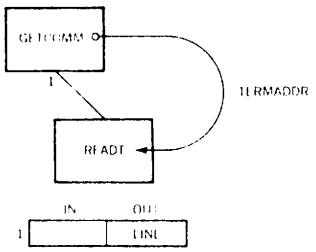
Typer af forbindelser (koblinger)

Forbindelser der adresserer til eller referer til et modul som helhed, ved dets navn, bevirker svagere kobling end forbindelser der refererer til interne elementer i et modul. Et eksempel på sidstnævnte tilfælde: Der bruges en variabel ved direkte at re-

ferere til den fra et sted inde i et andet modul. Ved fejlretning eller ændring skal hele dette andet moduls indhold evt. gennemgås for ikke at risikere uventede konsekvenser. Moduler der let kan bruges uden man egentlig kender deres indhold, gør systemer simplere.

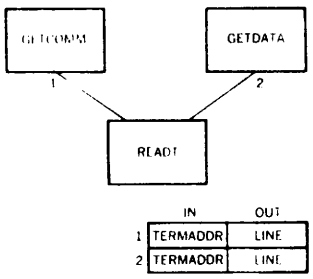
Betragt det der sker i figur 2. GETCOMM er et modul hvis funktion er at fremskaffe næste command fra en terminal. Under udførelsen af funktionen GETCOMM kaldes modulet READT, hvis funktion er at læse en linie fra en terminal. READT får denne linie via et externt declared data element i GETCOMM, kaldet TERMADDR. READT returnerer linien tilbage til GETCOMM, som et argument kaldet LINE. Bemærk pilen fra et sted inden i GETCOMM til et sted inden i READT. En pil af denne type beskriver referencer til interne data elementer i et andet modul.

Figur 2. Modul forbindelser



Nuvel, lad os forestille os at vi ønsker at tilføje et modul kaldet GETDATA, hvis funktion er at fremskaffe næste data linie (d.v.s. ikke en command) fra en (evt.) anden terminal. Det vil umiddelbart være fornuftigt at bruge READT som en subrutine til GETDATA. Men hvis GETDATA modificerer TERMADDR i GETCOMM før kald af READT, vil der opstå fejl i GETCOMM fordi der vil blive læst fra en anden terminal. Selv hvis GETDATA "restorer" TERMADDR efter brug, kan fejlen stadig opstå, hvis GETDATA og GETCOMM aktiveres samtidig i en multiprogrammerings installation. READT ville have været mere hensigtsmæssig, hvis TERMADDR havde været et input argument til READT, som vist i figur 3, i stedet for et externt declared data element.

Figur 3. Bedre modul forbindelser



Disse simple eksempler viser hvordan referencer til interne data elementer i andre moduler kan få utilsigtede konsekvenser, både med hensyn til økonomi og fejlmuligheder.

Typer af kommunikation

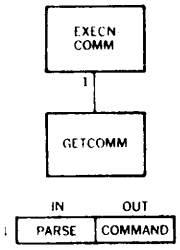
Moduler skal "som minimum" overføre data, ellers kan de funktionelt ikke fungere som en del af et system. Således, ved forbindelser, skal der som minimum udføres overførsel af data. (Dette gælder ikke ved kommunikationen af kontrol i et system. I princippet er tilstedeværelsen eller ikke af krævede input data nok til at definere de omstændigheder under hvilke et modul skal aktiveres, d.v.s. modtage kontrollen. Således, den klare og tydelige overgivelse af kontrol fra et modul til et andet,udgør en yderligere, teoretisk set uvæsentlig form for kobling. I praksis kræver systemer der udelukkende er "data-koblet" specielt sprog og operativsystem, men de er meget attraktive, ikke mindst fordi de fundamentalt er simplere end andre systemer af samme størrelse, der er "kontrolkoblet").

I teorien er det man bruger "harmløse", færrest mulig kontrolkoblinger ved subrutinecall's, det man kunne kalde overgivelse af "del-kontrol" eller et "kontrol-element".

Exempler på disse er en switch, en flag-byte eller anden form for signal fra et modul til et andet. Men egentlige kontrol argumenter er en yderligere komplicering af de nødvendige data argumenter der kræves for at eksekvere et job, og en alternativ mulighed der eliminerer denne komplicering er altid til stede.

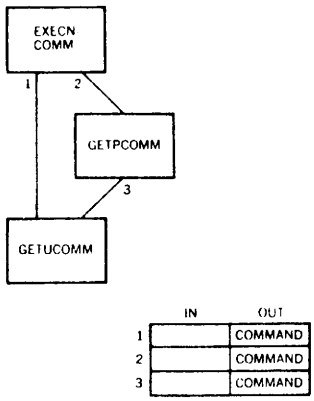
Betragt modulerne i figur 4.

Figur 4. Kontrol-koblede moduler



Modulerne er kontrol-koblede ved switch-en PARSE gennem hvilken EXECNCOMM "instruerer" GETCOMM om der skal returneres en bearbejdet (analyseret) eller ubearbejdet command. Opdeling af de to specifikke funktioner af GETCOMM resulterer i en struktur der er simplere, som vist i figur 5.

Figur 5. Simplificeret kobling



Det nye EXECNCOMM er nu ikke kompliceret; hvor det før satte en switch og kaldte, har det nu to forskellige call's. Sammenlagt er GETPCOMM og GETUCOMM (funktionelt) mindre kompliceret end GETCOMM var (p.g.a. mængden af switch-testning).Og de to små moduler er lettere at forstå end et stort. Indrømmet, de umiddelbare fordele opnået her, kan synes begrænsede, men de er stigende med tid og antal alternativer i switch'en. Kontrol kobling, hvor et kaldt modul "fortæller" det kaldende hvad det skal gøre, giver en strengere kobling.

Modificering af ét moduls kode ved hjælp af et andet modul kan opfattes som en sammenblanding af data og kontrol elementer; kodningen bliver opfattet som data af det modul der modificerer den, mens den optræder som kontrol i det modificerede modul.

Dette sidstnævnte modul gøres meget afhængig af det modificerende modul, og det modificerende modul bliver dybt involveret i andre modulers interne funktionering.

SAMMENHÆNG I ET MODUL (BINDING)

Kobling bliver svarede når relationerne til elementer der ligger uden for modulet minimeres. Der er to måder at opnå dette på. At minimere relationerne mellem modulerne, samt at "maximere" relationerne til elementer i samme modul. I praksis bliver begge metoder anvendt.

Den sidste metode er emnet for dette afsnit. "Element" i denne forbindelse er enhver form for "stump" af modulet, f.eks. et statement, et segment eller en "subfunktion". Hvis der er stærke interne bindinger i et modul, er der bedre sammenhæng i modulet. Målet her er at svække koblingen ("udadtil") ved at styrke interne bindinger. Skalaen af styrken af sammenhæng i et modul fra "svagest" til "stærkest" er som følger:

1. Tilfældig binding
2. Logisk "
3. Temporær "
4. Kommunikationsmæssig"
5. Sekventiel "
6. Funktionel "

Skalaen er ikke lineær, d.v.s. den er ikke stigende med et bestemt interval mellem de 6 trin. Funktionel binding er meget stærkere end nogen af de øvrige. Vi vil definere hver type af binding, og forsøge at indikere hvorfor den optræder på den givne plads i skalaen.

Tilfældig binding

Når der ikke er meningsfyldte, tilsigtede relationer mellem elementerne, "bestanddelene", i et modul, har vi den tilfældige binding. Tilfældig binding kan opstå ved en af følgende situationer: (1) Et eksisterende program modulopdeles ved at spiltte det af i moduler. (2) Moduler udarbejdes for at forhindre dobbelt-kodning i andre moduler. Et eksempel på de vanskeligheder der kan opstå ved tilfældige bindinger: Antag, at følgende sekvens af instruktioner optræder flere gange i et modul eller i flere moduler og derfor bliver lagt i et selvstændigt modul, kaldet X:

```
X: A = B + C;
   GET CARD;
   PUT OUTPUT;
   IF B = 4 THEN E = 0;
```

Modul X ville formentlig blive tilfældigt bundet, siden disse fire instruktioner ikke har åbenbare relationer mellem hverandre. Antag at vi i fremtiden i et af de moduler der oprindelig indholdt disse instruktioner, nu ønsker at sige GET TAPERECORD i stedet for GET CARD.
Se, det bliver et problem. Hvis vi ændrer i modul X, er det ubrugeligt for alle andre "kaldere" af X. Det ville måske oven i købet blive svært at finde alle kaldere af X for at lave en "kombinerbar" ændring.

Det skal med det samme indrømmes, at uafhængighed af modulets sammenhængningsgrad, er der tilfælde hvor ethvert modul kan ændres så det bliver ubrugeligt for "kalderne". Imidlertid, muligheden for dette er meget stor, hvis modulet indeholder tilfældige bindinger.

Logisk binding

Logisk binding,den næste i skalaen, vil sige at der på en eller anden måde er logiske relationer mellem elementerne i et modul. Eksempler er et modul der sørger for alle input eller output operationer i et program, eller editerer alle data i programmet.

Det logisk bundne, EDIT ALL DATA modul er ofte implementeret som følger: Antag, at de data elementer der skal editeres er master file record's opdateringer, sletninger og tilføjelser. Parametre der overføres til modulet vil bestå af data til editering og en speciel parameter der indikerer typen af data. De 4 første instruktioner er formentlig en form for "4-vejs hop", der hopper til 4 forskellige sektioner - edit master record, edit update record, edit deletion record og edit addition record.

Ofte er disse 4 funktioner "strikket sammen" på en eller anden måde i modulet. Hvis deletion record'en ændres, og dette kræver ændringer i "edit deletion record" funktionen, vil der opstå problemer hvis denne funktion er sammenstrikket med de 3 andre. Hvis de 4 edit funktioner er helt uafhængige af hinanden, kan systemet gøres simplere ved at lægge hver edit funktion i et separat modul. Så slipper man også for at tage beslutning om hvilken form for editering, der skal udføres ved hver exekvering.

Kort sagt, logisk binding resulterer ofte i "snirklet" eller delt "fælles" kode, som er svær at ændre og som kræver unødvendige parametre.

Temporær binding

Temporær binding er det samme som logisk binding, bortset fra at elementerne også er relateret i tid. D.v.s. at de temporært bundne elementer bliver exekveret i den samme tidsperiode.

De bedste exempler på moduler i denne "klasse" er traditionelle "initierings", "terminerings", "housekeeping" og "ryddeop"-moduler. Elementer i et initieringsmodul er logisk bundne, fordi initiering repræsenterer en "logisk kategori" af funktioner. Yderligere er disse elementer relateret i tid (nemlig på initieringstidspunktet).

Moduler der er temporært bundne synes at fremhæve ulemperne ved logisk bundne moduler. Imidlertid har temporært bundne moduler en tendens til at være simplere, af den grund, at alle elementer i et sådant modul er exekverbare på samme tid (d.v.s. der behøves hverken parametre eller logik for at bestemme hvilke elementer der skal exekveres). Derfor er de placeret højere på skalaen end logisk bundne moduler.

Kommunikationsmæssig binding

Et modul der er kommunikationsmæssigt bundet har elementer der er relateret ved referencer til det samme sæt af input og/eller output data. F.eks. "print and punch the output file", er kommunikationsmæssigt bundet. Denne form for binding er højere på skalaen end den temporære, fordi et modul med kommunikationsmæssig binding ligesom har "strengere forpligtelser" til at referere til de samme data.

Sekventiel binding

Når output data fra ét element er input til næste element, er modulet sekventielt bundet. Sekventielt bundne moduler kan være et resultat af at "flow-charte",et problem der skal løses, og derefter definere moduler der repræsenterer en eller flere blokke i flow-chartet. F.eks. "læs næste transaktion og opdater stamfilen" er sekventielt bundet.

Sekventielt binding er højt oppe på skalaen på grund af et tæt forhold til selve problemstrukturen. Men den er langt fra det maximale - funktionel binding. Grunden til dette er, at den "proceduremæssige" proces i et program er som regel forskellig fra

funktionerne i programmet. Derfor, et sekventielt bundet modul kan indeholde såvel flere funktioner som bare en del af en funktion. Dette resulterer som regel i strengere kobling, samt i moduler der sjældnere kan bruges af andre dele af systemet.

Funktionel binding

Funktionel binding er den stærkeste form for binding, I et funktionelt bundet modul er alle elementer relateret til udførelsen af én bestemt funktion.

Et spørgsmål der ofte rejses på dette punkt er: "Hvad er en funktion?". I matematikken Y=F(X) læses: Y er en funktion F af X. Funktionen F definerer en omdannelse af input variablen X til retur variablen Y. Altså, en funktion beskriver en omdannelse fra noget input data til noget retur data. I programmeringen har vi tilladt at denne definition også omfatter funktioner med ingen input data og funktioner med ingen retur data.

I praksis dækker ovennævnte definition ikke helt klart beskrivelsen af et funktionelt bundet modul. En rettesnor kan være, at hvis elementerne i modulet alle medvirker til opnåelsen af eet bestemt mål, er modulet formentlig funktionelt bundet. Eksempler på funktionelt bundne moduler: "Compute square root", (input og retur parametre), "Obtain Random number", (ingen input parameter) og "Write record on output file", (ingen retur parameter).

En nyttig teknik til bestemmelse af om et modul er funktionelt bundet er, at skrive en sætning der beskriver funktionen i (formålet med) modulet og derefter analysere denne sætning. Så kan man udføre følgende undersøgelser:

1. Hvis sætningen er nødt til at være sammensat, indeholder mere end ét verbum, udfører modulet formentlig mere end en funktion; derfor er det formentlig sekventielt eller kommunikationsmæssigt bundet.

2. Hvis sætningen indeholder ord der har relation til tid, som "første", "næste", "så", "efter", "når", "start" o.s.v. er modulet formentlig sekventielt eller temporært bundet.

3. Hvis man i sætningen, efter verbet ikke "opfatter" et enkelt, specifikt mål, er modulet formentlig logisk bundet. For eks. "Edit Source" har logisk binding; "Edit Source Statement" kan have funktionel

binding.

4. Ord som "initier", "rydde-op" o.s.v. medfører temporær binding.

Funktionelt bundne moduler kan altid beskrives via deres elementer med en sammensat sætning. Men hvis dette er uundgåeligt ved komplet beskrivelse af modulets funktion, er modulet formentlig ikke funktionelt bundet.

Et problem der står tilbage er, hvor længe man skal blive ved med at funktionsopdele. Opdelingen er formentlig færdig når ingen af modulerne indeholder et "subset" af elementer der kan bruges selvstændigt. Derudover skal modulet ikke være større end at hele dets indhold kan opfattes, "kaperes" på een gang, d.v.s. sjældent større end at sourcekoden kan stå på en eller to sider.

Bemærk at et modul kan have flere end én type binding. Bindingen mellem to elementer er den stærkeste der kan opstå. Bindingen i et modul gøres svagere, hver gang et par af elementer ikke fører til funktionel binding.

BRUGERVENLIGE MODULER

Et brugervenligt (brugeren er f.eks. vedligeholdelses-programmøren) eller "vel-tillavet" modul, er et, der når det får sit bestemte input, opererer identisk hver gang det exekveres. Det er også et, der opererer uafhængigt af sine omgivelser.

Denne karakteristik af brugervenlige moduler kunne evt. kaldes "black-box-ness". D.v.s. at brugeren kan forstå hvad modulet gør,og bruge det, uden egentlig at vide "hvad der er inden i". Modul "black-box-ness" kan yderligere opnås ved at kommentere moduler, så deres funktion gøres klarere. Ligeledes, et beskrivende navn, og veldefinerede, tydelige grænseflader gør modulet mere "black-box'et".

VIRKNINGER VED STRUKTURERET DESIGN

Omkostningerne ved at skrive mange simple moduler ligger i den exekveringstid og den lagerplads man bruger når man via det givne programmeringssprog udfører diverse call's. "Designeren" burde indse den modsatte effekt, den negative effekt, der opnås på vedligeholdelse og fejlfinding, når man stræber imod hurtig exekvering og/eller ringe brug af plads i maskinen. Han (hun) burde

også huske, at programmeringsprisen er måske ved at være den største udgift i et system, samt tænke på den fremtidige vedligeholdelse. Imidlertid, afhængig af omkostningerne ved programmeringssproget der bruges, er det meget sandsynligt at struktureret design kan resultere i lavere exekveringstid og/ eller mindre plads-krav, jvf. følgende overvejelser'

Angående lagerplads:

1. Sjældent brugte moduler bliver evt. slet ikke kaldt ind under exekveringen.
2. Struktureret design reducerer "dobbelt-kode" og reducerer kodning som er nødvendig for kontrol switches, altså en reducering af den totale programmør-genererede kodning.
3. Overlay strukturering kan baseres på aktuelle operative karakteristika, som indhentes ved kørsel og studier af programmet.
4. Når man har mange "single function" moduler har man mulighed for en mere flexibel, præcis gruppering, dette under enten overlay eller i en VS-installation.

Angående exekveringstid

1. Nogle moduler exekveres evt. kun få gange.
2. Sjældent brugte moduler kaldes evt. slet ikke ind og bruger derfor ingen exekveringstid.
3. Kodning til kontrol switches reduceres eller minimeres, reducerende den totale mængde kodning, der kan exekveres.
4. Meget ofte benyttede "grene" i systemet kan evt. rekompileres, hvorunder call's kan erstattes af "hop".
5. "Includes" eller "performs" (i COBOL) kan evt. bruges i stedet for call's. (Men det skal gøres med omtanke, det kan risikere at forøge komplexiteten).
6. En måde at opnå hurtig exekvering på er, at finde ud af hvilke dele af systemet der bruges mest, og så ofre den tid, der skal bruges til at optimere systemet på disse dele.Når man implementerer et strukturelt designet system, har man mulighed for at teste det med

henblik på at finde de "kritiske" moduler. Disse moduler kan så optimeres separat og re-integreres i systemet uden at forårsage fejl i dette.

STRUKTURERET DESIGN TEKNIKKER

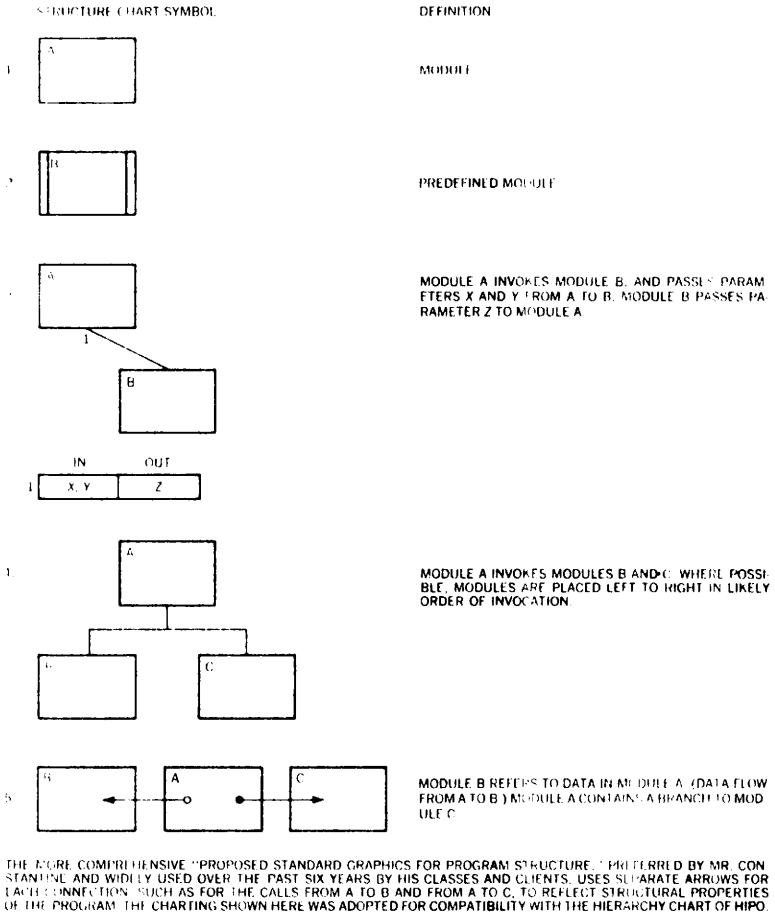
Det er muligt at opdele design processen i generel program design og detaljeret design som følger: Generel program design er at bestemme hvilke funktioner der skal bruges til dette program (eller programmeringssystem). Detaljeret program design er at bestemme hvordan disse funktioner skal implementeres. Ovennævnte overvejelser og betragtninger, og de teknikker der følger i dette afsnit, resulterer i en identificering af funktioner, parametre der skal overføres og de indbyrdes "call-forhold", der skal optræde i en struktur af funktionelt bundne, simpelt forbundne moduler. Informationerne der her fremkommer gør det lettere separat at designe, implementere og teste hvert modul.

Struktur diagrammer

Målet med den generelle programdesign er, at bestemme hvilke funktioner, parametre og "call-forhold" der er nødvendige.

Flow-charts beskriver hvornår (i hvilken orden og under hvilke forhold) blokkene exekveres, derfor komplicerer de unødigt den generelle program design fase. En bedre beskrivelsesmetode er et struktur diagram, som tidligere er beskrevet og som er vist i figur 6 - se side 15.

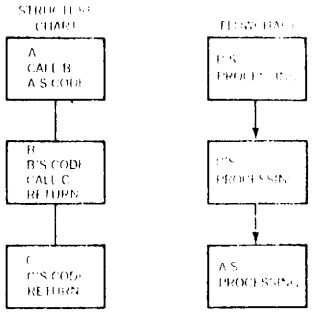
Figur 6. Definition af symboler i et struktur diagram



THE MORE COMPREHENSIVE "PROPOSED STANDARD GRAPHICS FOR PROGRAM STRUCTURE," PREFERRED BY MR. CONSTANTINE AND WIDELY USED OVER THE PAST SIX YEARS BY HIS CLASSES AND CLIENTS, USES SEPARATE ARROWS FOR EACH CONNECTION, SUCH AS FOR THE CALLS FROM A TO B AND FROM A TO C, TO REFLECT STRUCTURAL PROPERTIES OF THE PROGRAM. THE CHARTING SHOWN HERE WAS ADOPTED FOR COMPATIBILITY WITH THE HIERARCHY CHART OF HIPO.

For at sammenholde et struktur diagram og et flowchart, betragt følgende for de samme tre moduler i figur 7: A som kalder B, som kalder C.

Figur 7. Struktur diagram sammenholdt med flowchart



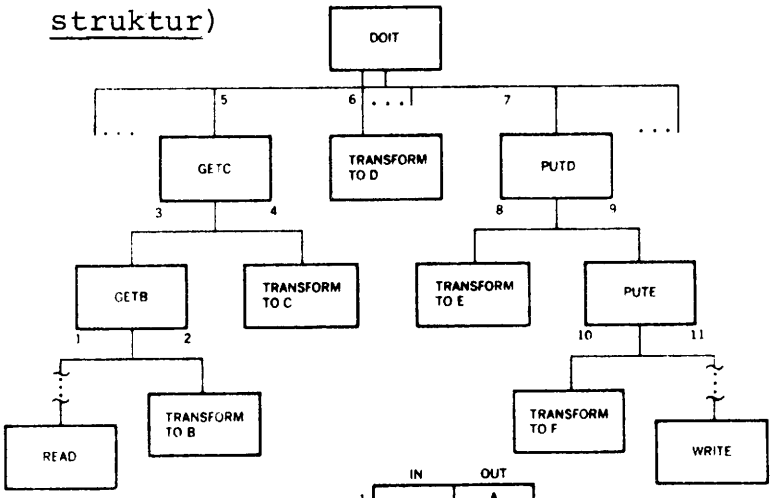
(For bedre sammenligning er kodningen beskrevet i struktur diagrammet; B's kode vil blive exekveret først, så C's, så A's). For at beskrive A's grænseflader korrekt, er det nødvendigt at vide, at A er "ansvarlig" for exekveringen af B. Det er svært at se ud fra flowchartet. Yderligere, struktur diagrammet kan vise modul forbindelserne og parameteroverførslerne,som er centrale i de overvejelser og teknikker der er præsenteret her.

Den anden hovedforskel der drastisk simplificerer beskrivelse og analyse under program design, er udeladelsen af "beslutningskassen" i struktur diagrammet. Betingede kald kan beskrives ved en sådan, men den egentlige beslutningstagen kan udskydes til den detaljerede modul design. Dette er et exempel på, hvordan design processen gøres simplere ved kun at beskæftige sig med en del af design problematikken ad gangen. Struktur diagrammer laves små nok til at "designeren" kan overskue det hele. Det modvirker "suboptimering" af dele af programmet, på bekostning af hoved-problemets løsning.

"Hoved-strukturer"

En genvej for at opnå simple strukturer er, at kende et generelt billede af resultatet af et system. L.L. Constantine har observeret, at programmer af den generelle struktur, vist i figur 8, resulterede i den billigste form for implementering.

Figur 8. Generel form for "lav-pris" implementering (transform-struktur)

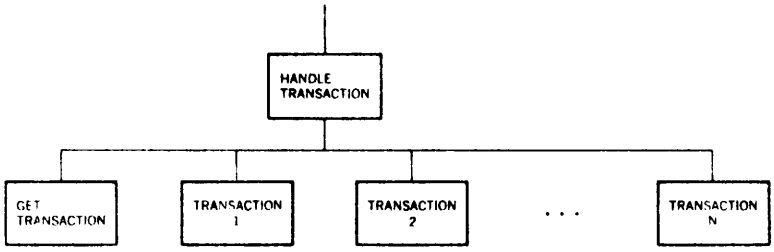


	IN	OUT
1		A
2	A	B
3		B
4	B	C
5		C
6	C	D
7	D	
8	D	E
9	E	
10	E	F
11	F	

Her vises input-process-output typen af et program, der appellerer til de fleste programmer, selvom "input" eller "output" er til "sekundært" eller internt lager.

En anden nyttig strukturering til brug ved implementering af dele af et design er en transaktions struktur, vist i figur 9.

Figur 9. Transaktions struktur



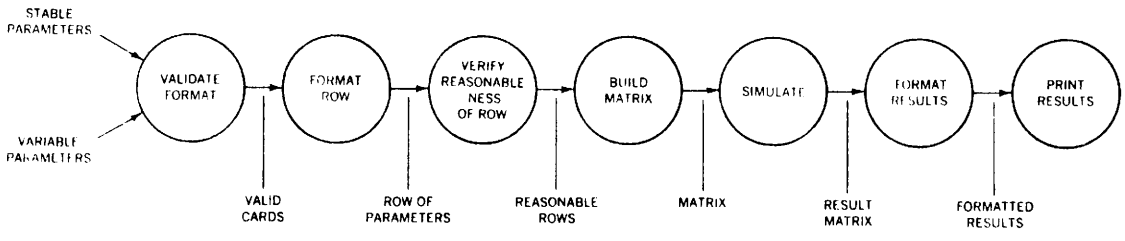
En "transaktion" her er et resultat, en record m.v. der resulterer i forskellige "aktiviteter". Strukturen kan optræde alene eller som et eller flere af "hoved"-modulerne (f.eks. GETC, PUTD eller PUTE i figur 8) i en input-proces-output struktur. F.eks. har en "command processor" (et system der f.eks. tager sig af udførelsen af operatørcommands på en dataskærm) denne struktur.

Design af strukturen

Følgende fremgangsmåde kan bruges til at konstruere input-proces-output strukturen i figur 8.

1. Første skridt er at skitsere et funktionelt billede af problemet. Som et eksempel antager vi, at der drejer sig om et simuleringssystem. En grov struktur af dette problem er vist i figur 10.

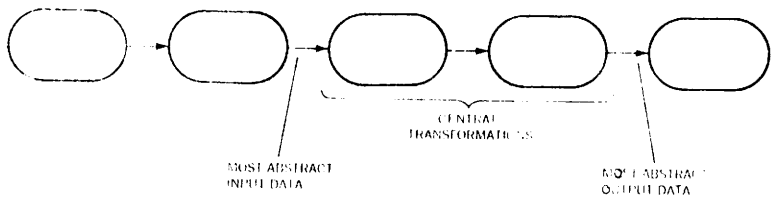
   Figur 10. Grov struktur af et simuleringssystem



2. Identificer de externe begrebsmæssige datastrømme. En extern datastrøm er en der er extern til systemet. En begrebsmæssig datastrøm er en relateret datastrøm, der er uafhængig af en hver fysisk I/O-enhed. F.eks., vi kan have flere begrebsmæssige datastrømme der kommer fra een I/O-enhed eller vi kan have een der kommer fra flere I/O-enheder. I vores simuleringssystem er de externe begrebsmæssige datastrømme input parametrene, og den formatterede simulering resultatet.

3. Identificer de vigtigste externe begrebsmæssige datastrømme (både input og output) i problemet - se figur 11. Derefter, under brug af problem-strukturen (figur 10) find ud af, hvornår disse vigtigste datastrømme er "længst væk" fra deres fysiske input form (mest abstrakte), men stadig kan betragtes som "kommende ind". Således, i simuleringssystemet er den mest "abstrakte" form af input transaktionsstrømmen måske "built matrix". Ligeledes identificer hvor datastrømmen først kan betragtes som "gående ud", i exemplet muligvis "result matrix".

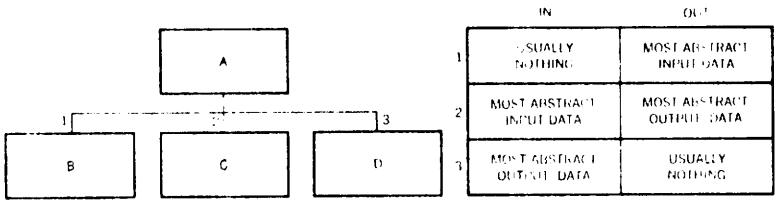
Figur 11. Bestemmelse af hvor data er fjernest fra fysisk input form.



Indrømmet, det er noget kryptisk noget. Erfaringerne viser imidlertid, at "designere", trænet i denne teknik, sjældent rammer ret meget ved siden af.

4. Lav strukturen i figur 12 udfra de hidtidige informationer. Lav et "over"-modul for hver input datastrøm der findes på det tidspunkt, hvor disse data er mest"abstrakte" (længst fra deres oprindelige form). Lav "under"-moduler på samme måde. Ofte er det kun nødvendigt med en enkelt "gren" mellem "over"- og "under"-modul. Parameteroverførslen er afhængig af problemet, men det generelle billede er vist i figur 12 - se side 19.

Figur 12. Højeste niveau



Beskriv funktionen af hver modul med en kort, præcis sætning. Beskriv hvilken transformering (omformning af data) der sker når modulet kaldes, ikke selve modulets indhold og arbejdsmåde. Vurder beskrivelsen i relation til funktionel binding.

Når modul A kaldes, exekveres systemet eller programmet. Således, funktionen af modul A er lig med det problem der skal løses. Hvis problemet er "lav en FORTRAN compiler", er modul A's funktion "kompiler FORTRAN program".

Modul B's funktion involverer behandling af "hoved"-datastrømmen. Exempel på "typisk modul B" er "get next valid source statement".

Modul C's formål er at transformere (omforme) "hoved"-input datastrømmen til "hoved"-output datastrøm. Dets funktion bør være en ikke-proceduremæssig beskrivelse af denne omformning.

Et exempel:

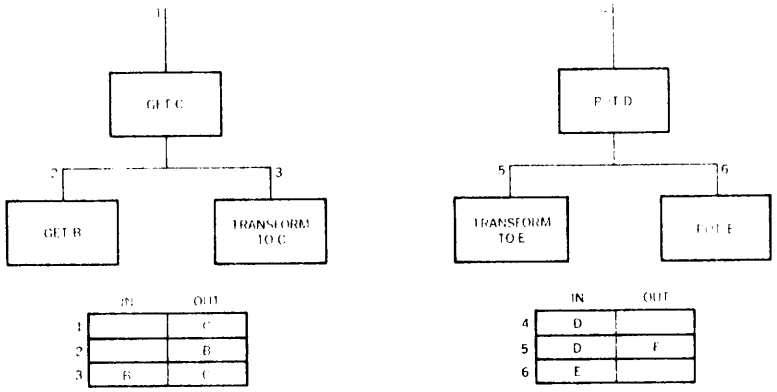
"Convert source statement to machine language statement".

Modul D's formål er at disponere hoved-output strømmen. Exempler er:

"Produce report", eller "display results of simulation".

5. For hvert "over"-modul, identificer den sidste transformering, der er nødvendig for at producere de data der returneres af modulet. Derefter, identificer udseendet (formen) af input umiddelbart før denne sidste transformering. For "under"-moduler, identificer den første proces, der er nødvendig for at "komme nærmere" det færdige output. Dette resulterer i dele af strukturen, som vist i figur 13.- se side 20.

Figur 13. Lavere niveauer



Gentag afsnit 5 indtil det oprindelige "hoved"-modul og de endelige "under"-moduler er nået. Modulerne kan analyseres i enhver rækkefølge, men hvert modul bør laves færdigt før nogle af dets underlagte moduler påbegyndes. Der er desværre ikke nogen detaljeret vejledning i at opdele i "transformeringsmoduler". Brug diverse overvejelser vedr. kobling og binding, størrelse (ca. een side source-kode), og brugbarhed (er der funktioner der kan bruges andetsteds nu eller i fremtiden).

I denne fase kan man komme til at underopdele for langt, men det er altid let at re-kombinere senere i design-fasen, men doublerende funktioner kan evt. overses, hvis opdelingen er for forsigtig her.

Design retningslinier

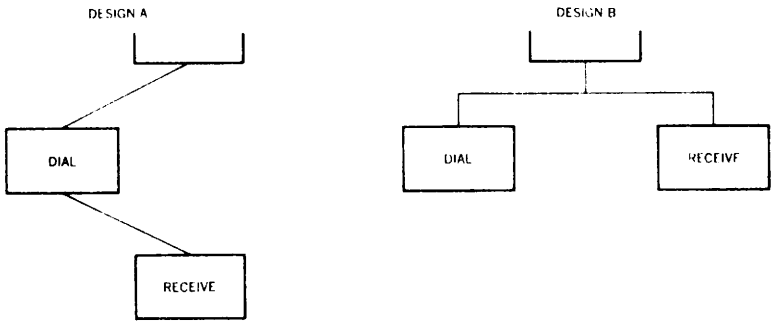
De følgende begreber er nyttige ved opnåelse af simple designs og "rigtige" strukturer.

Match program mod problem

En af de bedste teknikker til at undgå følgevirkninger af ændringer i programmet, er at lade strukturen af design'et "matche" med strukturen af problemet. F.eks. antag følgende: Vi har et modul, der "stiller ind" (dials) ud fra et telefonnummer der er drejet, og et der modtager data. Hvis modtagelse følger umiddelbart efter indstillingen, når man evt. frem til design A, vist i figur 14. Overvej imidlertid, om modtagelsen er en del af indstillingen. Siden den sædvanligvis ikke er det, så lad DIAL's

"caller" aktivere RECEIVE som i design B.

Figur 14. Design bør følge funktion

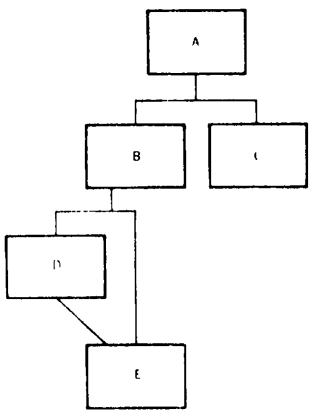


Hvis i dette exempel design A blev anvendt, overvej så følgen af et nyt krav om at transmittere efter indstilling, i stedet for at modtage. DIAL-modulet sørger for modtagelse, og kan ikke bruges. Så må man enten indføre en switch eller indføje et nyt DIAL-modul.

Rækkevidden (scope) af følgevirkninger og kontrol

Rækkevidden af kontrol (kontrolområde) af et modul, er modulet selv plus alle moduler der er underlagt det. I exemplet i figur 15 er B's kontrolområde lig med B,D og E. Rækkevidden af følgen af en beslutning (effektområde) er lig med hele "sættet" af moduler, der indeholder kodning hvis exekvering er baseret på resultatet af denne beslutning.
Det følgende exempel illustrerer hvorfor.

Figur 15. Rækkevidden af kontrol (kontrolområde)



Hvis exekveringen af noget kode i A er afhængig af udfaldet af beslutning X i modul B, har man to muligheder: Enten må B returnere et "signal" til A, eller også må beslutningen gentages i A. Det videre forløb resulterer i yderligere kodning for at indføje "signalet", eller også resulterer det i at noget af B's funktion (beslutning X) "ryger ud" i modul A. Doublering af beslutning X resulterer i vanskeligheder, hvis beslutning X skal ændres. Det kræver "koordinerende" ændringer i begge moduler.

Der er to muligheder for at bringe effektområdet indenfor kontrolområdet. Enten kan man bringe beslutnings-elementer "opad" i strukturen, eller også kan man flytte moduler der ligger inden for rækkevidden af følger, men uden for rækkevidden af kontrol "ind på plads".

Modul størrelse

Størrelse af moduler kan være en ledetråd, når der skal kigges efter potentielle fejl. Betragt omhyggeligt moduler med færre end 5, og flere end 100 exekverbare source statements. Moduler med meget få statements udfører måske ikke en hel funktion, d.v.s. er evt. ikke funtionelt bundet. Meget små moduler kan elimineres ved at placere deres statements i "kælderen". Store moduler udfører måske mere end én funktion. Et andet problem med store moduler er deres forståelighed og læsbarhed. Det er et bevist faktum, at man maximalt kan kapere 30 statements ved første gennemlæsning af et modul.

Fejl og end-of-file

Ofte er en del af et moduls funktion at gøre "kalderen" klart, at det ikke kan udføre sin funktion. Dette opnås ved at der returneres en fejl-parameter (helst binær). Et modul der behandler datastrømme må nødvendigvis kunne signalere end-of-file. Dette gøres også bedst med en binær parameter. Disse parametre bør imidlertid ikke fortælle kalderen hvad der skal gøres ved fejlen eller EOF. Ikke desto mindre, systemer kan gøres simplere hvis moduler kan designes uden brug af fejl-signaler.

Initiering

Ligeledes, mange moduler kræver at der udføres en eller anden form for initiering. Et initieringsmodul vil være "slapt" bundet, men et sådant vil af og til være den simpleste løsning.

Imidlertid vil det af og til være muligt at undvære brugen af initiering uden at gå på kompromis med "black box"-teorien (samme input producerer altid samme output). F.eks. et læse-modul får returneret en fejl for file-not-opened fra en access-metode. Det klares ved at åbne filen og læse igen. Derved elimineres nødvendigheden af initiering.

Selektér moduler

Undgå doblerende funktioner, men ikke nødvendigvis doblerende kodning. Det er en mægtig fordel, at når en funktion ændres, skal den kun ændres et sted. Hvis et modul synes næsten, men ikke helt at kunne bruges et andet sted i systemet, så prøv at identificere og isolere den brugbare del af funktionen. Resten af modulet kan så evt. indkobles i dets oprindelige "kalder".

Check moduler der har mange "kaldere", og moduler der kalder mange andre moduler. Det er ikke sikkert der er problemer, men der kan evt. mangle niveauer eller moduler.

Isolér afhængighed

Isolér alle "afhængigheder" i forbindelse med specifikke datatyper, records, index-strukturer m.v., til et modul eller til et minimum af moduler. Det modvirker følgevirkningerne af senere ændringer.

Reducér parametre

Kig efter muligheder for at reducere parameteroverførsler mellem moduler. Studér alle data der overføres som parametre for deres formål. Undgå at overføre hele records fra modul til modul, nøjes med at overføre det eller de felter der er nødvendige for at et givet modul kan udføre sin funktion. Ellers vil det være nødvendigt at ændre mange moduler, hvis f.eks. et felt udvides. Kun strengt nødvendige data-overførsler og strengt nødvendige fejl- og EOF-parametre bør være målet. Check binære switch'er hvad angår kontrolområde og effektområde.

Lad "designere" arbejde sammen, og lad dem gøre det om det komplette struktur diagram. De kan koordinere de forskellige grene af systemet som de hver for sig har arbejdet på, samt koordinere brugen af fælles moduler m.v.

ET EXEMPEL

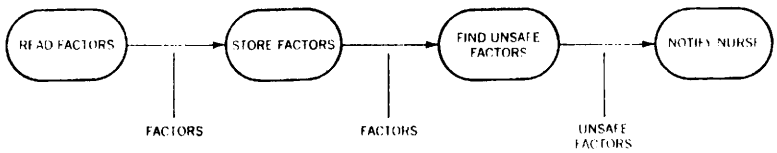
Det følgende exempel illustrerer brugen af struktureret design.

På et hospital skal der indføres et patientovervågningssystem. Hver patient overvåges via analogudstyr der måler faktorer som f.eks. puls, blodtryk og modstand i huden. Programmet læser disse faktorer periodisk (perioden spec. for hver patient) og lagrer dem i en data base. Sikkerhedsgrænser for hver patient er ligeldes registreret (f.eks. grænsen for patient X's temp. er fra 98 til 99,5 grader fahrenheit). Hvis en af faktorerne falder uden for patientens sikkerhedsgrænser, eller hvis en af enhederne "står af", signaleres der til sygeplejersken.

I "det pulveriserende liv" ville problemet indeholde langt flere detaljer, men dette problem er tilstrækkeligt detaljeret til at vi kan designe strukturen af programmet.

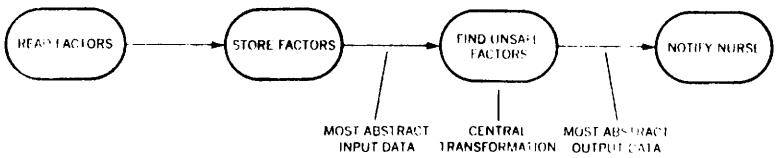
Første skridt er at lave en struktur af problemet, som vist i figur 16.

Figur 16. Problemstrukturens udseende



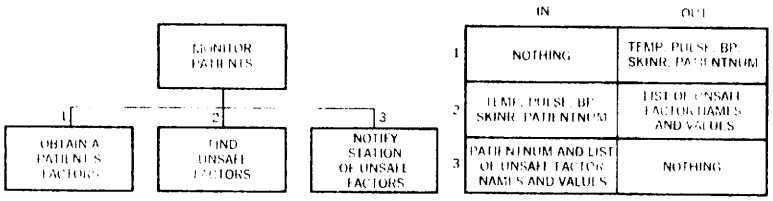
I det andet skridt identificerer vi de externe begrebsmæssige datastrømme. I dette tilfælde optræder der to datastrømme, nemlig faktorerne fra de analoge enheder og advarslerne til sygeplejersken. Disse repræsenterer også input- og output-"hoved"-strømme. Figur 17 indikerer det tidspunkt, hvor inputstrømmen er mest abstrakt (længst væk fra sin oprindelige form). Det er der, hvor patientens faktorer kan lagres på data basen. Output strømmen er mest abstrakt der, hvor strømmen er en liste af faktorer der overskrider sikkerhedsgrænserne (hvis der er nogle).

Figur 17. Mest abstrakte data



Nu kan vi begynde at designe programmets struktur, som vist i figur 18.

Figur 18. Strukturen på højeste niveau



I analysen af modulet "OBTAIN A PATIENT'S FACTORS", kan vi via problemformuleringen bestemme, at denne funktion har 3 dele:

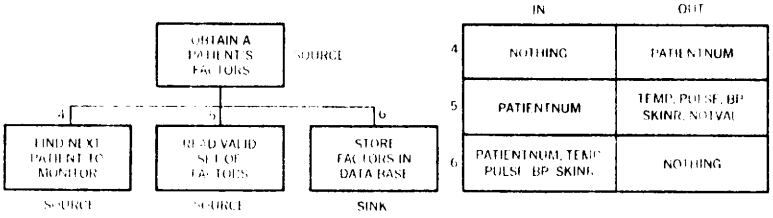
(1) Bestem hvilken patient der nu skal overvåges (ud fra det specifikke interval).

(2) Læs fra den analoge enhed.

(3) Placer faktorerne i data basen.

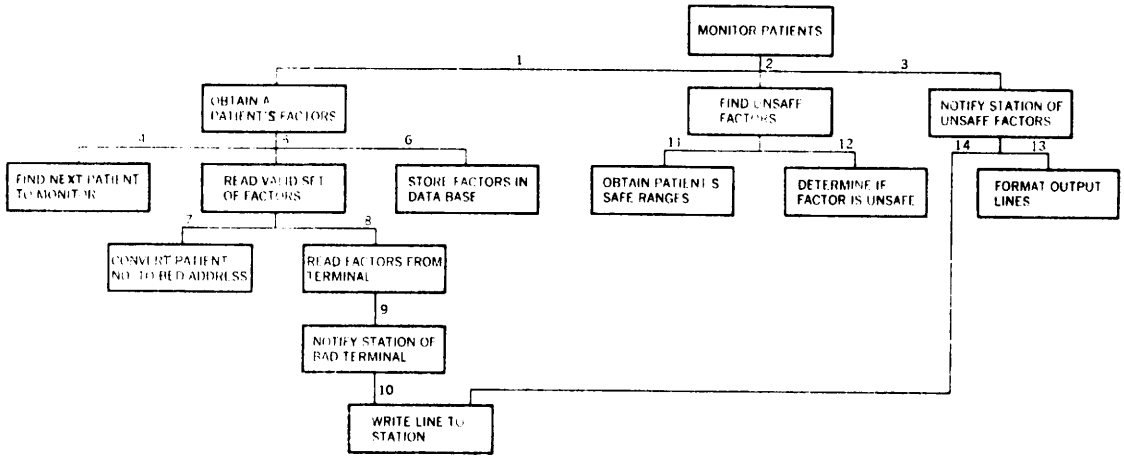
Altså, vi når frem til strukturen i figur 19. (NOTVAL bliver "sat", hvis et sæt valide faktorer ikke var til rådighed).

Figur 19. Struktur af næste niveau



Yderligere analyse af "READ VALID SET OF FACTORS", "FIND UNSAFE FACTORS", og "NOTIFY STATION OF UNSAFE FACTORS" bringer os frem til resultaterne som er vist i det komplette struktur diagram i figur 20 - se side 26.

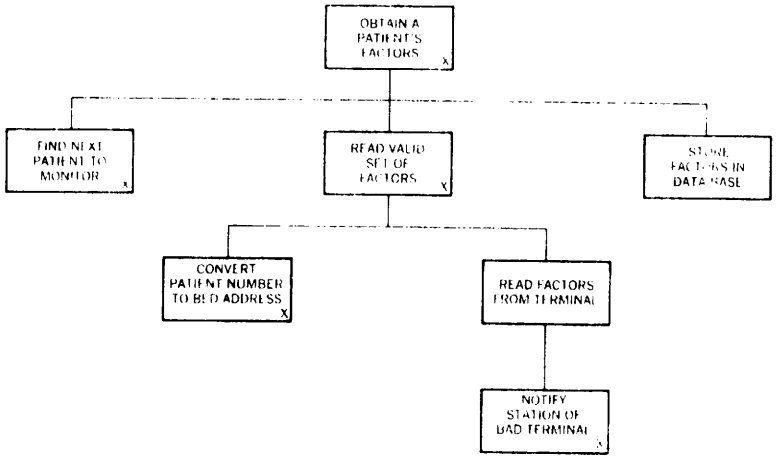
Figur 20. Komplet struktur diagram



	IN	OUT
1		TEMP, PULSE, BP, SKINR, PATIENTNUM
2	TEMP, PULSE, BP, SKINR, PATIENTNUM	LIST OF UNSAFE FACTOR NAMES AND VALUES
3	PATIENTNUM & LIST OF UNSAFE FACTOR NAMES & VALUES	
4		PATIENTNUM
5	PATIENTNUM	TEMP, PULSE, BP, SKINR, NOTVAL
6	PATIENTNUM, TEMP, PULSE, BP, SKINR	
7	PATIENTNUM	BEDNUM
8	BEDNUM	TEMP, PULSE, BP, SKINR, NOTVAL
9	BEDNUM	
10, 14	LINE	
11	PATIENTNUM	TEMPR, PULSER, BPR, SKINRR
12	FACTOR, RANGE	UNSAFE
13	LIST OF UNSAFE FACTOR NAMES AND VALUES	LIST OF LINES

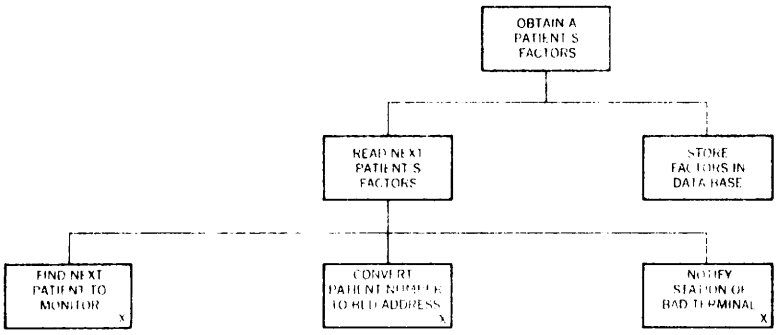
Bemærk at modulet "READ FACTORS FROM TERMINAL" indeholder en beslutning ud fra spørgsmålet "læste vi korrekt fra terminalen?". Hvis read'en indebar fejl, må vi signalere til sygeplejersken, og derefter finde næste patient der skal checkes, som vist i figur 21.

Figur 21. Strukturen som den er designet



Moduler der ligger indenfor effektområdet af denne beslutning er mærket med et X. Bemærk, at effektområdet ikke ligger indenfor kontrolområdet. For at korrigere dette problem må vi gøre det i to skridt. Først flytter vi beslutningen "op" til "READ VALID SET OF FACTORS". Det gør vi ved at tilføje "READ FACTORS FROM TERMINAL" ind i dets kaldende modul. Derefter lægger vi "FIND NEXT PATIENT TO MONITOR", som en "sub"-funktion til "READ VALID SET OF FACTORS". Derved når vi frem til strukturen i figur 22.

Figur 22. Effektområde indenfor kontrolområde



Altså, ved små ændringer dels i strukturen, dels i nogle få funktioner, har vi elimineret problemet.

KONKLUDERENDE BEMÆRKNINGER

Det hierakiske HIPO diagram bruges som hjælpeværktøj under den generelle system design. De betragtninger og teknikker der her er præsenteret, er nyttige, når man skal vurdere alternativer for de dele af systemet der skal programmeres til datamaskinen. Diagram-teknikken, der her er brugt, beskriver flere detaljer omkring grænsefladerne end det hierakiske HIPO diagram. Disse flere detaljer muliggør bedre betragtninger og overvejelser af hver individuel forbindelse og parameteroverførsel under den generelle program design. Det resulterende design kan dokumenteres med HIPO diagrammer. (Hvis designeren beslutter at have mere end en funktion i et givet modul, burde struktur diagrammet vise dem i samme blok. HIPO diagrammet derimod, vil vise alle funktioner i separate blokke). Outputtet af den generelle program design er input til den detaljerede modul design. HIPO input-proces-output diagram er nyttig

ved beskrivelse og design af hvert modul.

Struktureret design overvejelser kan bruges til gennemgang af program design's i et walk-through miljø. Disse begreber er også nyttige ved vurdering af alternativer for at opnå en-sides-program-segmenter i forbindelse med struktureret kodning.

Struktureret design reducerer den nødvendige indsats ved "reparation" og modificering af programmer. Hvis alle programmer var udviklet sådan, at der f.eks. var ét modul der henter records fra en master file, hvorfra vi får systemer, access-metoder, blokning af files eller I/O-enheder, ville disse programmer være meget forenklet. Og hvis alle programmer i installationen hentede records fra en given file ved hjælp af det samme modul, ville ét ændret modul løse problemer ved ændringer i file'n for alle programmerne i installationen.

Der er imidlertid andre fordele. Fejl undgås, når ens problem der skal løses er simplere. Hvert modul er meget selvstændigt og kan i nogen grad programmeres uafhængigt af andre ting som programmør, tid og programmeringssprog. Moduler kan testes inden al programmering er færdig ved at indføre simple "dummy"-moduler, der returnerer diverse "resultater" uden at "producere" dem. Kritiske moduler, hvad angår lagerplads og exekveringstid kan optimeres separat og re-integreres uden større besvær.

REFERENCER

1. L.L. Constantine, Fundamentals of Program Design, in preparation for publication by Prentice-Hall, Englewood Cliffs, New Jersey.

2. G.J. Myers, Composite Design: The Design of Modular Programs, Technical Report TR00.2406, IBM. Poughkeepsie, New York (January 29, 1973).

3. G.J. Myers, "Characteristics of composite design," Datamation 19, No. 9. 100-102 (September 1973).

4. G.J. Myers, Reliable Software through Composite Design, to be published Fall of 1974 by Mason and Lipscomb Publishers, New York, New York.

5. HIPO-Hierarchial Input-Process-Output documentation technique, Audio education package. Form No. SR20-9413, available through any IBM Branch Office.

6. F.T. Baker, "Chief programmer team management of production programming," IBM Systems Journal 11, No. 1. 56-73 (1972).

7. The use of the HIPO Hierarchy charting format is further illustrated in Figure 6, and its use in this paper was initiated by R. Ballow of the IBM Programming Productivity Techniques Department.

8. L.A. Belady and M. M. Lehman, Programming System Dynamies or the Metadynamics of System in Maintenance and Growth", RC3546, IBM Thomas J. Watson Research Center, Yorktown Heights, New York (1971).

9. L.L. Constantine, "Control of sequence and parallelism in modular programs," AFIPS Conference Proceeding, Spring Joint Computer Conference 32, 409 (1968).

10. G.M. Weinberg, PL/1 Programming: A Manual of Style, McGraw-Hill, New York, New York (1970).

11. Improved Programming Technologies: Management Overview,IBM Corporation, Data Processing Division, White Plains, New York (August 1973).