

## **Modulering af programmer**

Blokbegrebet og procedurebegrebet har hidtil været vejen til at fremstille meget store og meget komplicerede programmer. Mange højere programmeringssprog rummer faktisk ikke andre abstraktionsredskaber end disse to.

Ønsker man at lave programmer der benytter sig af avancerede datastrukturer slår procedurerne imidlertid ikke til. Man kan godt gøre det - mange programmer der benyttes i praksis er faktisk fremstillet alene ved hjælp af procedurer eller tilsvarende abstraktionsredskaber (fx FORMS i ABAP), men de resulterende programmer lader ofte noget tilbage at ønske med hensyn til læselighed og forståelighed.

Klassebegrebet er et abstraktionsredskab der søger at overkomme disse mangler ved at indføre nogle begrebsdannelser der gør det muligt at fremstille datalogiske modeller der "ligner" virkeligheden i højere grad end procedurerne tillader.

Med klasser til sin rådighed kan man fremstille datalogiske objekter der kan have egenskaber der afbilder egenskaberne ved objekter fra virkelighedens eller forestillingernes verden, og egentlig objektorienteret programmering bliver en mulighed. Samtidig giver den objektorienterede programmeringsstil mulighed for at indbygge kontrolforanstaltninger i programmer der skal bruges i mange forskellige sammenhænge, således at fejlagtig brug afsløres automatisk.

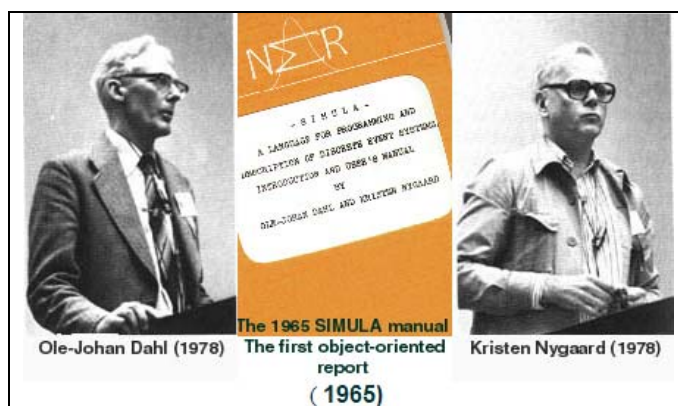
### **Historie**

I 1989 eksploderede interessen for objektorienteret programmering. Alle talte om det nye revolutionerende programmerings-koncept. Fra at være en kultagtigt form for programmering blev objektorienteret programmering noget alle talte om. Ord som arv, indkapsling og polymorfisme var på alles læber. De samme begreber blev brugt og måske især misbrugt i markedsføringen af mange produkter. Først i 1996 begyndte SAP AG at implementere objektorienteret tankegange i SAP R/3. I nogle år fandtes skjult under ALE konceptet abap objekter.

Mest kendt blev dog BAPI objekter som stod for Business Application Programming Interfaces. Ideen var god, fornuftig og realiserbar. SAP AG

tænkte abstrakt og indkapslede typisk bilag i forretningsmæssige objekter, hvor objektinterfacet blev gemt i et repository mens objektets metoder var implementeret som RFC funktioner. Rigtige objekter var der ikke tale om, men set udefra kunne objekterne anvendes igennem interfacet. Først med R/3 version 4.6 ser vi egentlig abap objects som programmeringssprog.

Programmeringens historien går fra ikke struktureret programmering (BASIC), over struktureret programmering (Pascal, Comal, C, Modula-2, PL/I, osv) til objekt-orienteret programmering. Objekt tanken optrådte første gang i 1965 i forbindelse med udviklingen af sproget SIMULA på norsk regnecentral. SIMULA blev udviklet som en overbygning på ALGOL. Det var til en helt speciel opgave, nemlig til simulering og optimering af busdriften i Oslo.



Opfinderne af den objektorienterede tankegang var Ole-Johan Dahl og Kristen Nygaard fra Oslo.

Siden hen var det en vigtig bestanddel af SMALLTAK udviklet tidligt i 1970'erne. I mange år herefter var det kun i Universitets kredse, at man studerede og anvendte objektorienterede sprog. Vi skal faktisk frem til 1990'erne førend de objektorienterede sprog for alvor slog igennem.

For Danmark har objektorienteret sprog en særlig betydning, idet Bjarne Strastrup stod bag C++ og Anders Hejlsberg stod bag Turbo Pascal (nu kendt som Delphi). Begge kommercielle programmeringssprog som udbredte den objektorienterede tankegang og teknik til de mange PC programmører. Dette er faktisk hovedårsagen til, at langt de fleste PC og WEB programmører idag har et glimrende kendskab til den

objektorienterede verden, selvom mange fortsat har svært ved at identificere og definere objekter.

I Mainframe verdenen, er den objektorienterede tankegang endnu ikke slået igennem.

De første udgaver af COBOL var nok næppe egentlig strukturerede sprog medens de seneste versioner byder på struktureret programmering og endda også mulighed for objekt-orienteret programmering. ABAP har fra starten været udviklet som et struktureret programmerings-sprog, dog med en stærk tilknytning til COBOL sproget.

Alle højniveau sprog giver mulighed for genbrug, i starten som include-filer senere som units og function modules.

Som struktureret sprog har ABAP hidtil anvendt forms, events, macroer, function modules, form pools, logiske databaser og include filer. Primært består byggestenene af forms og function modules, der giver programmøren overblik og dermed mulighed for genbrug. Fokus har således hidtil været på forms og function modules.

Mange af de design værktøjer og metoder vi hidtil har anvendt har enten taget udgangspunkt i data (DATAFLOW, ENTITETER) eller i funktioner (JACKSON). Med objekt-orienteret programmering og metoder flyttes grænserne, nu kombinere vi data og funktioner i samme objekter. Netop dette viser sig at være grænseoverskridende for mainframe programmører, der har svært ved at forestille sig at en "variabel" kan indeholde såvel data som funktioner.

ABAP Object blev introduceret i SAP R/3 4.0 og ses nu oftere og oftere i standard SAP. Når vi debugger kommer vil vi oftere komme til at skulle debugge noget objekt orienteret kode. Derfor er det vigtigt at forstå begreberne og i det midste kunne forstå hvad man ser i debug mode.

### **Objekter, hvorfor det**

Objekter kan bruges til at forenkle den virkelige verdens problemstillinger ved opbygning af abstrakte modeller, hvor vi vælger den detaljeringsgrad som vi ønsker at beskrive og forstå.

Hvis vi f.eks i en model af en problemstilling, har brug for at der indgår en kaffemaskine, kan vi vælge kun at beskrive det for problemstillingen relevante ved kaffemaskinen. Det kunne være hvorledes man laver kaffe. Hæld vand på til 6 personer, sæt kaffefilter i og 3 skefulde kaffe hældes ned i filteret hvorefter maskinen tændes. Lad os antage at det er det eneste der er relevant for vores problemstilling. Vi har dermed abstraheret fra alle de andre ting som kunne være interessant at vide om kaffemaskinen, alle de indre dele som bruger den 12V eller 110V, hvordan koger den vandet osv. Vi har her fat i noget essentielt for objekter/klasser, nemlig at vi som beskriver, definere og identificere objekter, har magten til at vælge vores abstraktionsgrad. Kunsten er så at vælge en passende abstraktionsgrad.

Vi er fra den virkelige verden, vant til at omgås og bruge objekter f.eks bil, cykle, bus, tog, færge, fly, hus, lejlighed, telefon, kaffemaskine og meget andet.

I den virkelige verden benytter vi os ofte også af stor abstraktion, idet vi næsten alle formår at anvende en telefon uden at vi forstår telefonens virkemåde. De fleste kan også nemt lære at betjene en bil, uden de af den grund skal kunne forklare hvorledes bilens motor virker og får bilen til at køre frem. Objekter er et begreb vi ofte bruger om ting, hvorfor vi finder det lettere at abstrahere over dem og blot bruge dem.

Tænk f.eks på lego klodser som objekter eller som komponenter der kan sammensættes på mange måder og i forskellige forbindelser og alligevel stammer de alle fra én bestemt klods.

Det behagelige ved et enkelt objekt er ofte dets størrelse eller dets begrænsede funktionalitet, hvis blot vi kender objektets formål og ved hvad vi skal aflevere til objektet og hvad vi kan forvente som resultat, så er vi ikke bange for at bruge objektet i forskellige sammenhænge.

Den som har udviklet objektet har også en stor chance for at kunne teste objektet 100%, idet det enkelte objekts kompleksitet ikke er større end at den kan gennemskues. Endelig vil den som har udviklet objektet, kunne forandre objektets indre kompleksitet, f.eks performance optimere, uden at det iøvrigt kræver recompilering af de programmer som anvender objektet.

I mange år har man forsøgt at lave modeller ud fra enten funktioner eller data. Vi kender alle til SYSKON diagrammer, JSP, Yourdon, Dataflow diagrammer, funktionsdiagrammer og entitetsdiagrammer. Med objekter får vi mulighed for at kombinere data og funktioner og kan derved nemmere håndtere modeller. Når vi samtidigt kan indkapsle kompleksiteten bliver vi også i stand til at vælge abstraktions niveau.

En af de hyppigst anvendte argumenter for at anvende objekt-orienteret sprog er at kvaliteten af programmerne vil blive forbedret. Men hvad ligger der bag dette "kvalitets" begreb:

I den forbindelse nævnes normalt:

**Korrekthed**, at programmet gør det som det skal ifølge specifikationerne.

**Robusthed**, at programmet også kan fungere under abnorme betingelser.

**Udvidbarhed**, at programmet ret nemt vil kunne tilpasses nye betingelser og krav.

**Genbrug**, at dele af eller hele programmet vil kunne genbruges.

**Kompatibilitet**, at programmet kan snakke sammen med andre programmer.

Det er især blevet fremhævet, at programmer skrevet indenfor en objekt-orienteret ramme har fordele m.h.t udvidbarhed og genbrug og mere generelt omkring vedligeholdelse, som efterhånden udgør 70% af software omkostningerne. Ofte kan der spares mere end 30% af kildekoden med objekt-orienteret programmering. Det fremhæves også at fejlprocenten i programmer skrevet med objekter er væsentlig mindre end i traditionelle programmer.

Dette skyldes især objekternes størrelse og det at man under analyse- og design har fået fastlagt ind- og uddata mere præcist. Det er derfor lettere at teste mindre kodefragmenter og sikrer, at de leverer de rigtige informationer ud fra de modtagne data og instruktioner. Vi kan også tidligere under analyse og design, teste vores objekter og designe kommunikationsflowet mellem objekter og mellem objekter og det klassiske program.

## Basale begreber

Sammen med objekt-orienteret programmering dukkede der en række nye begreber op. Desværre anvender de forskellige objekt-orienterede sprog ikke de samme begreber. I dette dokument anvendes ABAP begreberne. Der er 4 hoved begreber som uværligt er knyttet til den objektorienterede verden og som skal ligge på ryggraden.



### Abstraction/Abstraktion

Med abstraktion menes at vi gennem definition af klasser har mulighed for at kunne modellere alverdens problemer uden at gå i detaljer. Vi kan altså arbejde med klasser uden at kende en classes implementation blot ved at kende klassens publicerede grænseflade.

### Encapsulation / Indkapsling

Indkapsling af data og metoder. Med dette mener vi at attributter, felter eller properties hører sammen med et sæt af metoder eller funktioner. Med indkapsling opnår vi, at kunne skjule objektets indre kompleksitet og kun publicere de metoder og attributter vi ønsker at stille til rådighed for omverdenen.

### Inheritance / Arv

Med arv mener vi genbrug. Vi nedarver egenskaberne fra en anden klasse har derefter mulighed for at tilføje vores egen klasse nogle nye egenskaber som properties og metoder. Det er altså reelt genbrug og ikke en klon eller kopi af noget sourcekode.

### Polymorphisme / Flertydighed / Overstyret

Med polymorphisme menes at vi ved nedarvning af en anden klasse kan reimplementere en nedarvet metode, altså omskrive metodens

funktionalitet dog ikke metodens interface. Vi har altså mulighed for at genbruge og alligevel tilpasse enkelte metoder til vores behov. Polymorphisme bruges ofte i constructor metoder og ofte i den situation hvor vi har brug for at tillægge funktionalitet eller attributter.

### Vigtige begreber

Ud over de basale begreber kommer her en række begreber som er nødvendige at kende.

#### Klasse

En af de måder hvorpå moderne programmeringssprog adskiller sig fra ældre sprog som fx Fortran, er disse sprogs faciliteter for gruppering af data i datastrukturer. Med datastruktureringsredskaber til rådighed bliver det muligt at designe programmer ud fra et mere totalt synspunkt: fra design af procedurer, dvs. fokusering på algoritmerne, over imod organisering af data.

Der opstår et programmeringsparadigme der går ud på at gruppere procedurer og de data de manipulerer i en form for moduler, hvor den faktiske implementation skal være usynlig for brugeren. En sådan konstruktion kaldes en **abstrakt datastruktur**.

Objekter defineres i abap som værende en klasse, ganske som i C++. En klasse indkapsler objektets data og metoder. Under en klasse erklæres objektets komponenter. Alle komponenter skal tilknyttes information om hvorvidt komponenten er synlig "public" eller ikke synlig "private" eller beskyttet "protected". Vi kan altså betragte en klasse som værende en abstrakt beskrivelse af et objekt eller bygge tegningerne til objektet.

Klassens komponenter opdeles i visibilitets/synlighed:

- Public
- Protected
- Private

Som komponenter i ABAP Object kan der erklæres:

#### Instans komponenter

Data  
Methods  
Events

#### Static components

Class-Data  
Class-Methods  
Class-Events  
Types  
Constants

#### Interfaces

Interfaces  
Aliases

Forskellen mellem klasser og objekter er at klassen eksistere på design tidspunktet hvor objektet kun lever på runtime tidspunktet. Det er i vores klasse vi definere properties og metoder og efterfølgende implementere i vores program. Objektet opstår først på runtime tidspunktet når det instantieres.

#### Objekt

Objektet er vores variabel eller instans af klassen, som på runtime tidspunktet indeholder klassens properties og metoder.

#### Instans

Vi høre/læser om instans variabler, at vi instatiere, om instans komponenter mm. At instantiere vil sige at vi initialisere. En instans variabel opstår når vi opretter et objekt med kommandoen CREATE OBJECT. Derfor taler vi også om en eller flere instanser af en eller samme klasse. Her kunne vi også have sagt at vi har flere variabler med samme definition nemlig defineret som en klasse.

#### Class Data / Class Method / Class Event

Med disse begreber menes data, metoder og events som er generelle for klassen og som optræder som globale for hele klassen og dermed kun én gang for alle instanser af klassen = objekter.

#### Instance Data/ Instance Method / Instance Event

Med disse begreber menes data, metoder og events som er unike for objektet og som optræder som lokale for objektet og dermed pr. Instans af klassen = objekt.

#### Definition

Definitionsdel af en klasse er der hvor vi definere eller publisere vores klasse overfor omverdenen. Vi beskriver klassens public, private og protected data og metoder. Det svarer således til en beskrivelse af en datatype. Man kan også sige at klassen er den konstruktions tegning som ligger til grund for fremstilling af objektet.

## Implementation

Implementationsdelen af en klasse er der hvor vi har programmeringen af den enkelte metode. I implementationen finder vi såvel classic som objektorienteret abap kode.

## Objekter

Der er mange aspekter som skal overvejes når man taler om objekt orienteret tankemåde. Generelt kan der være tale om forretningsmæssige objekter eller tekniske objekter. De forretningsmæssige objekter fremkommer igennem analyse og design af et system eller problem område, hvor tekniske objekter fremkommer gennem trinvis forfinelse af programmer.

### Forretningsmæssige objekter

De forretningsmæssige objekter identificeres ofte i forbindelse med kortlægning af en virksomheds processer og data entiteter. Objekterne vil typisk være virksomhedens faste begreber som f.eks kunde, leverandør, medarbejder, ordre, faktura. Begreber som er generelle for flere virksomheder og begreber som er unikke for en enkelt virksomhed. Nogle af disse begreber dækker over intressenter og identificeres af en interessentanalyse, andre er relateret til dokumenter som ordre, faktura, girokort etc. For hver af disse begreber bør man spørge sig selv hvilke informationer man har brug for ~ properties og hvad man skal kunne gøre med begrebet ~ metoder.

De forretningsmæssige objekter behøver ikke nødvendigvis programmeres objektorienteret, men det er hensigtsmæssigt at "tænke" objekt orienteret. Har man f.eks et forretningsmæssigt besluttet, at alle virksomhedens kunder skal være oprettet i en egen udviklet "skygge" tabel, som anvendes til synkronisering med et legacy system, vil det være hensigtsmæssigt, at tænke på at udvikle nogle generelle metoder til sikring og kontrol af at denne synkronisering er foretaget.

Implementeringsmæssigt kan dette jo løses via funktionsmoduler, user exits og change pointers. Men hvis vi forretningsmæssigt tænker i objekter, begreber og metoder, tilsikres at vi får udviklet generelle funktionsmoduler som kan kaldes i stedet for at mange programmer, indeholder samme eller næsten samme kode.

Formålet med at tænke på forretningsmæssige objekter er, at sikre en høj grad af genbrug, at identificere de generelle funktionsmoduler man får brug for i en organisation, til at vedligeholde og kontrollere sine forretningsmæssige begreber.

Da det er vanskeligt at tænke objekt orienteret og specielt vanskeligt at forholde sig til forretningsmæssige objekter vil vi i det efterfølgende koncentrere os om de tekniske objekter.

### Tekniske objekter

Tekniske objekter er for programmøren noget nemmere at identificere og forstå. I PC'ens barndom skulle alle programmører vide noget om katodestrålens vandring hen over skærmen for at undgå 'sne' på skærmen. Hvem tænker over dette idag. De tekniske objekter opstår ofte fordi programmøren gentagne gange har skrevet samme eller næsten samme kode for at løse et problem. Derfor forsøger man at generalisere problemstillingen og løser det med udvikling af funktionsmoduler eller klasser. Her er klasse eller objekt begrebet blot en yderligere trinvis forfinelse.

### Hvad er et objekt?

Her kalder vi det et objekt og ikke en klasse fordi jeg igennem denne beskrivelse beskriver instanser altså virkelige objekter som lever på runtime tidspunktet.

Normalt er objekter håndgribelige ting i vores omgivelser – fra papir clips til biler, busser, tog mm. De to ting som alle objekter har fælles er deres adfærd og tilstand.

Et objekts adfærd kan beskrives ud fra hvordan objektet reagere og påvirkes. Hvis vi for eksempel sparker til et objekt, så vil objektet clips flyve igennem luften, men sparker vi til en person vil denne sikkert sparke igen. Sparkes der på en saks vil det resultere i snitsår og sparkes der til en bus risikeres et brækket ben. Tilstanden af et objekt kan måles som sted, vægt, farve og hastighed. Tilstanden kan repræsenteres som data og adfærd som programmer, rutiner eller metoder.

Objekter består af adfærd, tilstande og deres indbyrdes kombinationer.

I starten kan det være endog meget vanskeligt at identificere objekter, og det til trods for at vi dagligt er omgivet af mange objekter og forstår at anvende disse objekter uden at vi tænker nærmere over det.

Objekter kan både være fysiske og abstrakte

Fysiske objekter:

- Dankort automat
- Mobil telefon
- Cykel
- Bil
- Bus
- Fly
- PC
- Regnemaskine
- Kaffemaskine

Abstrakte objekter:

- Kunde
- Person
- Skærm
- Menu

I foil præsentationen viste jeg en række objekter og vi blev klar over at vi er alle vant til at være omgivet af objekter, faktisk har vi siden barndommen automatisk brugt den objekt orienterede synsvinkel. Når børn ser et billede af et objekt kan de genkende og beskrive hvad objektet bruges til. Vi genkender objekter alene udfra billedet eller en beskrivelse og tænker ikke på hvordan objektets indre fungerer, vi taler her om **INDKAPSLING**.

Men vi er med det samme klar over hvordan et objekt betjenes. Ofte vil vi udfra en fornemmelse eller en betjenings vejledning være i stand til at anvende selv komplicerede objekter. Dette gør vi naturligt og uden at tænke på, objektets indre kompleksitet, vi udviser **ABSTRAKTION**. De fleste mennesker er i stand til at betjene en mobil telefon eller et video kamera, uden at være i stand til at kunne beskrive virkemåden. Ser vi f.eks en stjernebillede, vil vi automatisk vide hvordan det betjenes men kun de færreste ved hvordan billedet er konstrueret.

Det at vi kan tænke og behandle objekter som billeder, gør at vi hurtigt får et overblik og nemt kan træffe beslutninger. Skulle vi i stedet tænke over alle objekternes konstruktion og virkemåde, hver gang vi så et billede, ville vi ikke kunne

behandle de mange daglige synsindtryk.  
(ABSTRAKTION)

Alle objekter har to fælles træk, de har en adfærd og en tilstand. Adfærden er objektets reaktion på anvendelsen af objektets betjeningsflader. Objektets tilstande er målbare f.eks vægt, størrelse, farve, hastighed, alder mm. Tilstande repræsenteres i objektet som værende data. Adfærd repræsenteres i objektet som værende metoder.

Før i tiden var det meget normalt at designe edb systemer udfra en analyse af tilstande og aktioner, hvor man via tilstands diagrammer dannede sig et overblik over det gamle "logiske" system og herefter forandrede det til det nye logiske system.

Senere blev den mest populære model Yourdon's dataflow diagrammer, hvor man afdækker datastrømme i edb systemet.

Database specialister ynder at anvende relationsdiagrammer, der tager udgangspunkt i en dataanalyse der giver stor viden om forretningen men ikke afdækker funktioner. EDB systemer designet af DB specialister har typisk programmer pr. entitet.

Med blokbegrebet og procedurebegrebet som de eneste abstraktionsredskaber, er mange store og komplekse datasystemer blevet udviklet. Men programmerne lader meget tilbage at ønske med hensyn til læsevenlighed og forståelighed.

I den objektorienterede verden designes systemet objektorienteret, f.eks udfra systemets begreber, interessenter, væsentlige entiteter etc. Altså dermed systemets data abstraktion også kaldet klasser. De objektorienterede sprog giver mulighed for i højere grad at simulere den virkelige verden.

Data og funktioner er samlet. Den objektorienterede verden analyserer data og funktioner samtidigt. Typisk starter man med at identificere de abstrakte data og finder dernæst ud af hvad man skal kunne gøre med et objekt hvorved funktioner afdækkes.

Et objekt kunne være en salgsordre. Enhver salgsordre består af nogle data som adressat, identifikation, leveringsadresse,

leveringsbetingelser, generelle oplysninger, varelinie med antal, enhedspris, pris, moms oplysninger og totalpris. Endvidere vil der ofte være andre former for information. Objektet "Salgsordre" indeholder således en række attributter (data) vi kalder også disse data for properties.

En salgsordre opstår jo ikke bare uden at nogen eller noget har igangsat funktioner, vi skal derfor definere hvad vi skal kunne gøre med en salgsordre:

- Oprette salgsordre
- Rette salgsordre
- Slette salgsordre
- Vise salgsordre
- Udskrive salgsordre

Med andre ord har vi her identificeret salgsordrens metoder. Indkapslingen består i, at anvenderen af objektet kan nøjes med at læses objekt beskrivelsen og her få at vide hvilke informationer objektet stiller til rådighed igennem hvilke metoder (funktioner).

I objektbeskrivelsen (klassen) defineres såvel interne som eksterne kendte data og metoder. De abstrakte aspekter findes i klassen hvor de virkelige aspekter findes i objektet på runtime tidspunktet.

Når vi skal identificere objekter eller klasser af objekter hjælper det at identificere interessenter af systemet, ting, tabeller, bilag og virksomhedens vigtige begrebs apparat. Vi skelner mellem forretnings relaterede objekter/klasser og tekniske objekter/klasser.

**Public components**, er attributter, metoder og hændelser som kan ses og direkte adresseres af en bruger eller et vilk. Program.

**Protected components**, er attributter, metoder og hændelser kan adresseres af klassens objecter og subklasser, men ikke af andre.

**Private components**, er attributter, metoder og hændelser som kun kan ses og adresseres direkte internt i objektet.

Public, protected og private er klassens Visibility sektioner, altså en definition og opdeling af data og metoder.

Målet for objekter er at være i stand til at sikre deres egen konsistens. De fleste af et objekts data er derfor interne og dermed erklæret som private

Udskrevet: 15-02-2010

data. Private attributter kan kun forandres gennem objektets egne metoder. Som hovedregel har et objekt kun metoder defineret som public. Dermed kan objektet sikre egen konsistens.

Vi ser dog også objekter som publicerer visse attributter, vi kender det fra mange microsoft produkter eks. Microsoft Access, Visual Basic, Excel etc. Der har publiceret en række "Properties". Objekter har også en entydigt identifikation for at adskille dem fra andre objekter der har samme metoder og attributter.

## Class Pools

Introducere en Class Pool skrevet I ABAP kildekode. CLASS-POOL kommandoen skal være den første kommando i kildekoden. Class Pools oprettes og vedligeholdes via CLASS BUILDER i ABAP WORKBENCH. Hovedprogrammer som anvender en klasse får automatisk indsat et include statement til at inkludere class-pool. Men husk at Class Pools kun kan vedligeholdes via CLASS BUILDER.

## Interface Pools

Introducere en Interface Pool skrevet I ABAP kildekode. INTERFACE-POOL kommandoen skal være den første kommando i kildekoden. Interface Pools oprettes og vedligeholdes via CLASS BUILDER i ABAP WORKBENCH. Hovedprogrammer som anvender et interface får automatisk indsat et include statement til at inkludere interface-pool. Men husk at Interface Pools kun kan vedligeholdes via CLASS BUILDER.

## Data Objects

Et dataobjekt er en instans af datatypen og optager lige så meget hukommelse som specificeret I datatypen. Oprettes via kommandoen CREATE DATA.

Med data objekter, er det nu muligt at erklære variabler uden angivelse af datatypen som først er kendt på runtime tidspunktet.

Eks.

CREATE DATA myref TYPE (c). "hvor C er en variabel indeholdende datatypen
---

Behovet for data objekter er opstået i forbindelse med klasser, idet klassens datatyper jo først er

kendt på tidspunktet for oprettelse af instans variabler.

```
DATA:          dref      TYPE REF TO
DATA.
TYPES:          booking TYPE sbook.
FIELD-SYMBOLS: <fs>    TYPE ANY.

CREATE DATA dref TYPE booking.
ASSIGN dref->* TO <fs>.
```

I ovenstående eksempel erklæres dref som værende en reference til noget data, altså ikke feltet direkte, men en pointer til datatypen. <fs> erklæres som værende en pointer til data felter af vilkårlig type. Med CREATE DATA fortæles at dref har datatypen sbook. Den efterfølgende assign til <fs> medføre at <fs> nu har datatypen sbook. Dette kan virke omstændigt, men giver mulighed for at erklære variabler dynamisk på runtime tidspunktet.

→ Jeg behandler ikke data objects yderligere, men slå op i online hjælpen, her findes eksempler som umiddelbart kan afprøves.

## Object referencer

```
INTERFACE il.
..
ENDINTERFACE.

CLASS cl DEFINITION.
  PUBLIC SECTION.
    INTERFACES il.
  ENDCLASS.

TYPES iref TYPE REF TO il,
       cref TYPE REF TO cl,
       dref TYPE REF TO iref.
```

Hvis du specificere en global eller lokal class, dannes data typen for en class reference variabel og den statiske type i den specificerede klasse. Sådanne reference variabler kan pege på alle instanser af klassen og subclasser. Generelt kan du bruge en klasse reference til at adressere alle synlige komponenter af et objekt. Det samme gør sig gældende for interfaces.

## ABAP Objects

### CLASS

Klasser kan oprettes globalt via CLASS BUILDER eller lokalt i ABAP programmer, her koncentrerer vi os alene om lokale klasser defineret i ABAP programmer.

CLASS kommandoen introducere en kodeblok som afsluttes af ENDCLASS.

Man kan ikke erklære klasser in procedurer (subrutiner, funktionsmoduler eller under metoder) og man kan ikke erklære klasser inde i klasser. Som andre globale data, bør klasser defineres i begyndelsen af et program.

I definitions delen af en klasse erklæres klassens komponenter. Alle komponenter skal tilhøre en synligheds sektion. Komponenter kan være **public**, **protected**, eller **private**. Deklations delen af en klasse opdeles derfor i 3 sektioner:

#### [PUBLIC SECTION](#)

Indeholder komponenter som er tilgængelig for alle.

#### [PROTECTED SECTION](#)

Indeholder de komponenter som kun er tilgængelig indenfor klassen eller dennes arvinger.

#### [PRIVATE SECTION](#)

Indeholder de komponenter som kun er tilgængelig indenfor klassen.

Da en klasse ikke har en default synlighed skal deklarationen starte med en sektion, ydermere skal disse sections angives i en bestemt rækkefølge.

Indenfor hver sektion kan man erklære følgende komponenter:

#### Instans komponenter:

##### [DATA](#)

Instans attributer

##### [METHODS](#)

Instans metoder

##### [EVENTS](#)

Instans hændelser

#### Statiske komponenter:

##### [CLASS-DATA](#)

Statiske attributer

##### [CLASS-METHODS](#)

Statiske metoder

##### [CLASS-EVENTS](#)

Statiske hændelser

## TYPES

Interne typer anvendt i klasse

## CONSTANTS

Statiske konstanter

## **Interfaces:**

### INTERFACES

for komponenter fra interfaces

### ALIASES

for alias navne på interface  
komponenter

Når vi taler om instans komponenter er det komponenter som først findes når et objekt eller instans variabel oprettes. Først når kommandoen CREATE OBJECT er udført, vil instans komponenter eksistere. For en given klasse vil der findes ligeså mange instansvariabler af et givent klasse felt som der findes antal af instanser.

Statiske komponenter er knyttet til klassen og findes derfor allerede lige efter erklæringen af klassen. De statiske komponenter findes kun én gang pr. Klasse.

Interfaces implementere interfaces til klassen, de kan betragtes som en udvidelse af PUBLIC SECTION.

Man kan ikke anvende lokale datatyper i en klasse, du kan kun anvende globale synlige datatyper, dem som findes i datadictionary eller dem som er erklæret før definitionen af din klasse.

## **DEFERRED**

Hvis du har flere klasser erklæret i dit program og de forskellige klasser refererer til hinanden skal du huske at udvide din klasse definition med kommandoen **DEFERRED** hvilket blot betyder at du viser at klassen er defineret på forhånd (altså før implementeringsdelen)

## **PUBLIC**

Definere en global klasse i CLASS BUILDER. Du kan ikke bruge denne kommando i et abap program.

## **INHERITING FROM superclass**

Anvendes når man erklære en klasse hvor man ønsker at arve egenskaberne fra en anden klasse ofte kaldet superklasse eller forældre. Den nye klasse arver alle data og metoder fra superklassen, men kun komponenter fra PUBLIC SECTION og

PROTECTED SECTION er tilgængelige fra den nye klasse.

En klasse kan kun have netop én superklasse som forældre men kan selv være superklasse for flere subclasses.

## **Konklusion**

Skulle man med meget få ord forsøge at beskrive hvad ABAP OBJECTS handler om, så må det være noget retning af.

### ***Forenkling af kompleksitet med Indkapsling & abstraktion.***

*Det vanskeligste ved den objektorienterede verden, er nok det at håndtere de mange abstrakte begreber og design løsninger. Vi har det alle med helst at kende til alle detaljer, før vi kaster os ud i en løsning. I objekternes verden skal vi tænke helt anderledes. Undgå detaljer, indkapsle og skjul detaljerne mest muligt og brug ekstra tid på at gennemtænke snitfladerne til metoder og klasser. Ved design af klasser skal man anvende mindst to synsvinkler, nemlig designerens syn **TOP-DOWN** hvor systemet nedbrydes og programmørernes **BUTTON-UP** hvor programmørerne udnytter genbrug af eksisterende løsninger. Den perfekte balance findes et sted mellem disse to grundlæggende forskellige synsvinkler.*

*Når man starter med at anvende klasser, behøver man ikke forsøge at bygge en helt ny verden, husk de to synsvinkler top-down og button-up. Som programmør bør du bare tænke på, at kontrollere om SAP ikke har en klasse som løser netop dit behov og hvis ikke så led efter funktionsmoduler.*

*I top-down behøver klasser ikke være perfekte fra start, klasser har den gode egenskab at de kan udvikle sig hen af vejen. Hvergang en klasse lærer en ny metode, vil denne metode komme alle andre anvendere af klassen tilgode. Hvis en gammel kendt metode forandres, vil denne forandring slå igennem til alle de programmer der anvender klassen.*

*Der er ingen undskyldninger - klasser er simpelt hen bare fantastiske.....!*