10
Nedarvning I.

Udvidelse og specialisering.

Klassehierarkier.

Nedarvningsterminologi.

Interfaces.

Statiske og dynamiske typer.

Polymorfi.

Dynamisk binding og virtuelle operationer.

Decentraliseret/centraliseret software arkitektur.

Abstrakte klasser.

Nedarvning og genbrug.

PS1 -- Nedarvning I

Noter

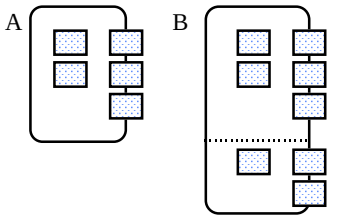
I denne forelæsning introduceres nedarvning og de vigtige begreber og mekanismer, der er knyttet til nedarvning. Denne forelæsning knytter sig ikke til et specielt sprog. Specielt er denne forelæsning ikke knyttet til Eiffel.

I næste forelæsning vil vi se på, hvordan nedarvning understøttes i Eiffel, og vi vil se på nogle eksempel-programmer programmeret i Eiffel.

I den tredie og sidste forelæsning om nedarvning vil vi se på, hvordan nedarvning og specifikation med logiske udtryk hænger sammen. Endvidere vil vi i denne forelæsning diskutere nedarvning fra mere end én klasse.

Udvidelse og specialisering (I).

- Der er ofte behov for at *udvide* en abstrakt datatype med nye egenskaber eller
- at *redefinere* eksisterende egenskaber til nye egenskaber.



- B har alle A's egenskaber, plus nogle nye.
- Vi ønsker ikke i B at skulle gentage eksisterende egenskaber fra A.
- Vi ønsker at fastholde, at der eksisterer objekter, der kun har A's egenskaber.
- A og B operationer skal tilgås og anvendes på samme måde i instanser af B.
- *B implementeres derfor som en ny klasse, der arver fra klassen A.*

- Udvidelse fører til specialisering af typen.
  - Objekter af typen B er også af typen A.
  - Objekter af typen B er mere specifikke og detaljerede end objekter af typen A.

PS1 -- Nedarvning I

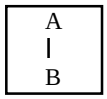
Noter

Som det diskuteres i 10.2.3 af OOSC afspejler udvidelse og specialisering to forskellige perspektiver. Når vi taler om udvidelse, refererer vi de nye egenskaber, vi tillægger B. Når vi taler om specialisering tænker vi på den delmængde af objekter, der kan siges at have disse udvidede (eller ændrede) egenskaber.

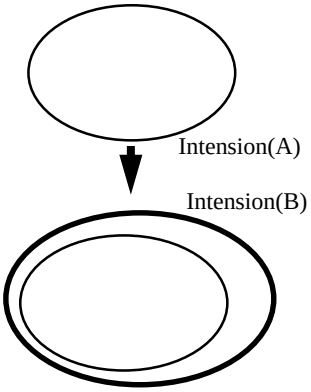
Denne slide udtrykker nogle helt centrale ønsker til udvidelses- og specialiseringsmekanismen i objekt-orienterede programmeringssprog.

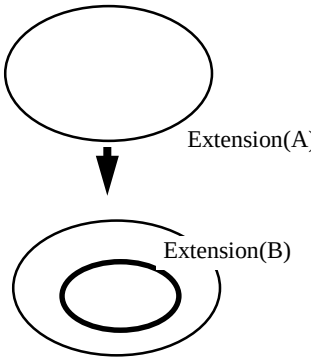
Som vi vil se på næste slide, afspejles disse to syn i hhv. intension og ekstension. Disse begreber blev allerede introduceret i forelæsningen om abstrakte datatyper.

Udvidelse og specialisering (II).



En klasse B arver fra A





Intensionen
Tilføjelse af nye egenskaber
Udvidelse

Extensionen
Indsnævring af begrebet
Specialisering

PS1 -- Nedarvning I

Noter

Lad os minde om definitionerne på intension og extension:

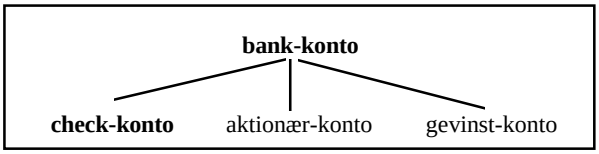
Intension : karakteristiske, påkrævede egenskaber.

Extension : mængden af fænomener, som er dækket (beskrevet) af begrebet.

Den **fede** oval betegner hhv. intension(B) og extension(B).

Her og i det følgende vil vi grafisk illustrere at Barver fra A ved den lille figur i øverste højre hjørne. Bemært at relationen mellem A og B ikke er symmetrisk på nogen måde. I stedet for at have en pil på linien mellem A og B benytter vi konsekvent konventionen at superklasser (dem vi arver fra) er foroven, og subklasser (dem der arver) er nedenunder.

Eksempel: udvidelse og specialisering.



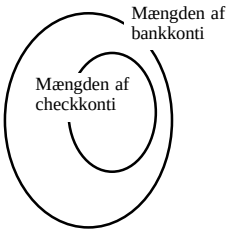
Egenskaber
(Intension)

Bankkonto: Kontoid
Indestående
Ejer
Rentesats

\+

Checkkonto: Antal frie checks
Indløs-check!

Objekter
(Extension)



PS1 -- Nedarvning I

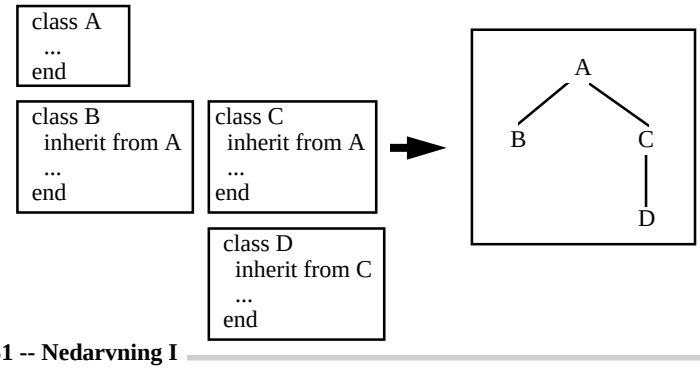
Noter

En instans af en checkkonto har alle bankkontoegenskaberne og alle de mere specifikke chekkontoegenskaber. Det er vigtigt at bide mærke i, at i et objekt af typen checkkonto, er disse egenskaber ligeværdige, og de kan bruges på samme måde. I en klient af checkkonto skal man således ikke tænke på, om denne eller hin egenskab oprindelig er programmeret i selve klassen checkkonto, i bankkonto, eller måske i en evt. superklasse til bankkonto.

Disse observationer afspejler noget helt centralt i den objekt-orienterede programmeringsstil.

Klasser, nedarvning og klassehierarkier.

- Syntaktisk udvides en klasse, som arver fra en anden klasse, med angivelse af en eller flere superklasser.
  - Én superklasse: *enkelt nedarvning* (single inheritance).
  - Flere superklasser: *multipel nedarvning* (multiple inheritance).
- Angivelse af superklasser i klasse-definitioner definerer et klassehierarki:



PS1 -- Nedarvning I

Noter

Nedarvning af egenskaber fra en klasse A til en klasse B er et programmeringssprogligt modstykke til specialisering af begrebet A til B. Vi siger ofte, at vi specialiserer klassen A, når vi angiver, at A er en superklasse til klassen B; og vi siger, at der er et specialiserings/generaliseringsforhold mellem klasserne A og B.

Tilsvarende er sammenhængen mellem en klasse og instanser af klassen et modstykke til klassificering/eksemplificering mellem begreber og fænomener.

Hvis en klasse B har en attribut (variabel) af typen A, kan en instans af B gennem attributnavnet referere til en instans af klassen A. Dette definerer, som bekendt, en anden relation mellem klasser: klasse-klient forholdet.

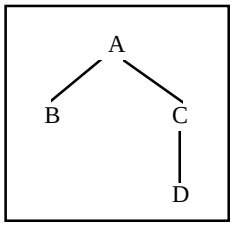
Det er vigtigt at holde disse sammenhænge mellem klasse-klasse og klasse-instans helt klar.

Der henvises til forelæsningen om abstrakte datatyper, hvor nogle af ovennævnte "ord" er defineret og beskrevet.

Opgave: Diskuter muligheden af at indføre generaliserings-specialiserings forholdet mellem klasser via klasse-klient forholdet. Mere konkret: Vi ønsker, at A er en superklasse til B. Kan vi arrangere dette ved i B at erklære en variabel x af typen A, således at vi stadig opnår en form for nedarvning af variable og routiner?

Instantiering og initialisering af klasser. med nedarvning.

En klasse B, der arver fra en anden klasse A, instantieres som ét objekt, der har alle egenskaber fra B og A.

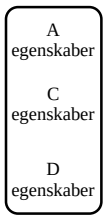


Initialisering af instanser af en subklasse.

Problem:

Hvordan kombineres initialiserings-operationerne i subklasse og superklasser?

En instans af klassen D



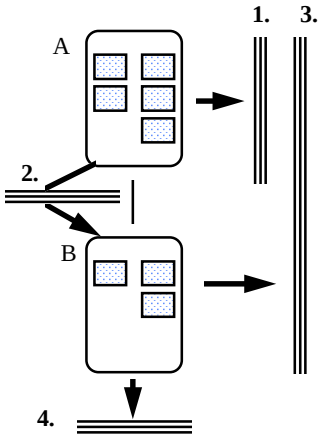
PS1 -- Nedarvning I

Noter

Vi nøjes på denne slide med at identificere initialiseringsproblemet af attributter i superklassen. I næste forelæsning vil vi se på, hvordan det er løst i Eiffel.

Interfaces til klienter og og subklasser.

B er en subklasse af A:



Interfaces til klienter:

1. De egenskaber i A, som klienter af A kan anvende.

3. De egenskaber i A + B, som klienter af B kan anvende.

Interfaces mellem sub/superklasser:

2. De egenskaber i A, som operationer i B kan anvende.

4. De egenskaber i A+B, som operationer i subklasser kan anvende.

PS1 -- Nedarvning I

Noter

På denne slide benytter vi notationen A+B for klassen B for at understrege, at B har egenskaberne i A foruden sine egne.

Som omtalt i en tidligere forelæsning er en klasse's interface den mængde af klassens egenskaber, som er synlig og brugbar udaftil i forhold til klassen. I princippet kan de fire illustrerede interfaces defineres helt uafhængigt af hinanden.

I alle praktiske sammenhænge, og i kendte objekt-orienterede sprog, er der dog en kobling mellem de forskellige interfaces. Koblingen kan f.eks. være, at 1 er en delmængde af 2, eller at 1 er en delmængde af 3.

Interfaces af typen 2 og 4 afspejler egenskaber defineret i superklasser, som er tilgængelige i subklasser. I nogle sprog er dette interface identisk med mængden af egenskaber i superklassen. Det er, med andre ord, ikke muligt i sådanne sprog at have noget privatliv i en klasse i forhold til superklasser.

Der er meget stor variation fra sprog til sprog mht. til de faciliteter, der stilles til rådighed for definition af interfaces.

Statiske og dynamiske typer.

Den *statiske type* af en variabel eller parameter er den type, hvoraf variablen eller parameteren er erklæret.

Den *dynamiske type* af en variabel eller parameter er typen af det objekt, variablen eller parameteren refererer til.

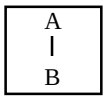
```
X: A;          X har statisk type A
Y: B;          Y har statisk type B

X.create(...); X har dynamisk type A
Y.create(...); Y har dynamisk type B

--hvis lovligt:
X := Y;        X har dynamisk type B
Y := X         Y har dynamisk type B
```

```
class A
  ...
end --A
```

```
class B
  inherit A
  ...
end --B
```



PS1 -- Nedarvning I

Noter

Skelnen mellem statisk og dynamisk type af en variabel eller parameter er meget vigtig for at kunne forstå det nært beslægtede emne: statiske kontra dynamisk binding og virtuelle operationer. Vi vender tilbage til dette på efterfølgende slides.

Det er også muligt og nyttigt at tale om den statiske hhv. den dynamiske type af et udtryk. Via typeerklæringer af variable, operatorer og funktioners resultat kan man udtale sig om et udtryks statiske type.

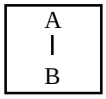
På næste slide vil vi definere under hvilke omstændigheder de to assignmentets nederst på denne slide er lovlige.

Polymorfi.

"Polymorfi": mangeformethed

Variable og parametre kan referere til objekter af mere end én type.

Lad B være en subklasse af A.



***Typiske regler* for assignment og operationskald:**

v := e — Antag, at v har statisk type A: v: A.
Assignmentet er lovligt i typemæssig forstand hvis den statiske type af e er en klasse, der er en subklasse til A.

f(e) — Antag at f er en procedure eller funtion defineret som f(p: A).
Kaldet er lovligt i typemæssig forstand hvis den statiske type af e er en klasse, der er en subklasse til A.

F.eks. lovlige, hvis e har statisk type B.

PS1 -- Nedarvning I

Noter

Polymorfi betyder som omtalt "mangeformethed". Med lidt andre ord kan man tænke på polymorfi som det at kunne "have eller antage forskellige former".

Man kan naturligvis ikke give universelle regler for assignments og operationskald på tværs af programmeringssprog. De viste regler udmærker sig ved, at de kan checkes på det tidspunkt programmet oversættes. Årsagen til dette er, at der kun indgår statiske typer i reglerne. Dette er en stor fordel, idet man så undgår relativt kostbare checks på programmets udførelsestidspunkt.

Lad os vende tilbage til de to assignments X := Y og Y:= X fra forrige slide.

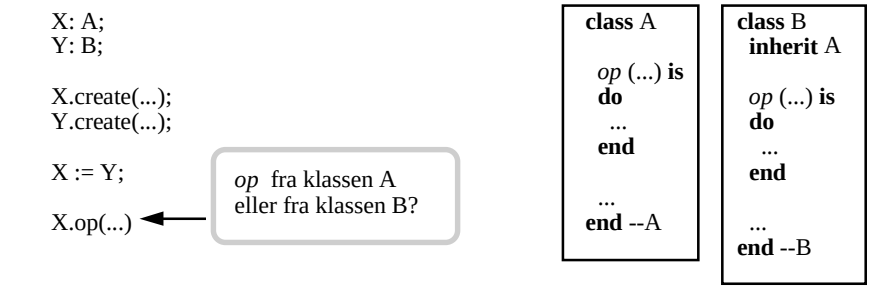
Ifølge reglen indført på denne slide er X := Y lovlig, idet den statiske type af Y (nemlig B) er en subklasse af den statiske type af X (nemlig A).

Igen ifølge reglen er Y := X ulovlig. Årsagen er, at X´s statiske type (A) ikke er en subklasse af Y´s statiske type (B).

Når de to assignments forekommer i den viste rækkefølge, kan det føles restriktivt ikke at tillade Y:= X. Forskellige sprog har forskellige konventioner på dette punkt. Nogle sprog tillader Y := X, og disse sprog følger således ikke de regler, vi har formuleret på denne slides. Sådanne sprog må ty til typecheck på programmets udførelsestidspunkt.

I forelæsningen "Nedarvning II" vender vi tilbage til disse ting (se slide "Typesammenlignelighed og reverse assignment attempt").

Statisk og dynamisk binding.



Dynamisk binding: Den dynamiske type af X er bestemmende for bindingen af *op* .

Statisk binding: Den statiske type af X er bestemmende for bindingen af *op* .

PS1 -- Nedarvning I

Noter

Hvis der anvendes statisk binding, kaldes op fra klassen A i X.op(...).

Omvendt, hvis der anvendes dynamisk binding, kaldes op fra den specialiserede klasse B i X.op(...).

I sprog med statiske typer (i programteksten) er det vigtigt, at anvendelse af dynamisk binding ikke svækker typesikkerheden. Sproget må derfor gennem de sproglige regler sikre, at uanset den dynamiske type af X, findes der en operation op, der kan anvendes.

Man må ikke forveksle betegnelserne 'statisk binding' og 'dynamisk binding', som anvendt i objekt-orienteret sammenhæng, med 'statisk navnebinding' og 'dynamisk navnebinding', som defineret i forelæsningerne om grundlæggende begreber i programmeringssprog. (Sidstnævnte handler om, efter hvilken disciplin frie navne i en konstruktion bindes).

Dynamisk binding og virtuelle operationer.

Problem:

> I hvilke situationer skal der anvendes dynamisk binding, og i hvilke skal der anvendes statisk binding?

Forskellige løsninger:

Programmør-bestemt.
Egenskab ved operationer:

> Operationer, der "binder dynamisk" defineres som **virtuelle.**

Sprog-bestemt

> Sproget anvender dynamisk (eller statisk) binding i alle situationer.

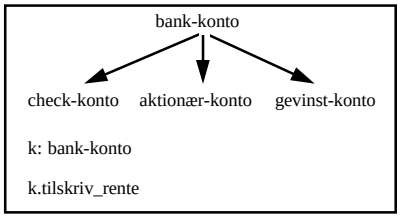
Noter

Årsagen til, at vi har sat statisk binding i parantes nederst på sliden, er, at hvis der helt firkantet og én gang for alle skal vælges mellem statisk binding og dynamisk binding, så er dynamisk binding langt den mest nyttige og interessante.

I den næste forelæsning, som primært handler om hvordan nedarvning finder sted i Eiffel, vil vi vende tilbage til, hvordan forskellige sprog takler dette problem.

Decentraliseret softwarearkitektur.

```
procedure tilskriv_renter(k: bankkonto)
begin
  if k.type = checkkonto
   then tilskriv_rente_checkkonto(k)
   elsif k.type = aktionærkonto
   then tilskriv_rente_aktionærkonto(k)
   elsif k.type = gevinstkonto
   then tilskriv_rente_gevinstkonto(k)
   else ...
end
```



Der er behov for eksplicit type-diskrimination i programmet.

Hver transaktionstype pr. operation kræver sit eget navn.

Introduktion af en ny kontotype kræver globale ændringer i programmet.

Introduktion af ny transaktion kræver lokale ændringer i programmet.

Den dynamiske type af k bestemmer implicit, hvilken rentetilskrivningsform, der anvendes.

Det samme operationsnavn kan anvendes på alle kontotyper.

Introduktion af en ny kontotype kræver lokale ændringer i programmet.

Introduktion af ny transaktion kræver globale ændringer i programmet.

PS1 -- Nedarvning I

Noter

Det antages, at der findes en operation *tilskriv_rente* på alle typer af bankkonti i kontohierarkiet.

Løsningen til venstre er en mulig konventionel løsning, givet vi ønsker at have een abstraktion, der tilskriver rente.

Løsningen til højre er den naturlige objekt-orienterede løsning, som er centreret om dataabstraktion.

Som man kan se, er den objekt-orienterede løsning "god", hvis man tilføjer en ny kontotype; den anden løsning er "god", hvis man tilføjer en ny operation på tværs af alle kontotyper.

En "god" løsning i denne sammenhæng er karakteriseret ved, at man kan nøjes med at ændre lokalt for at gennemføre ændringen.

I den objekt-orienterede løsning må man ændre globalt, hvis det kræves, at nye operationer defines på de enkelte kontotyper. Hvis en ny operation på kontotyperne kan udtrykkes ved hjælp af de allerede eksisterende operationer på kontotyperne, er den objekt-orienterede løsning dog stadigvæk "god".

Abstrakte klasser (I).

En klasse er en **abstrakt klasse**, hvis der findes operationer i klassen med en ikke-defineret krop.

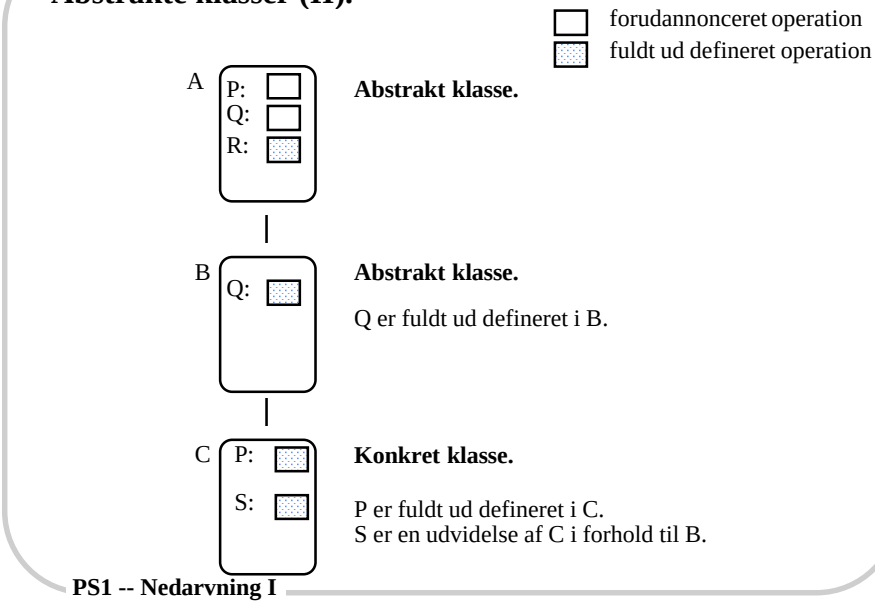
Sådanne operationer kan siges at være *forudannoncerede.*

- Forudannoncerede operationer i en abstrakt klasse A skal defineres i subklasser af A.
- Overordnede egenskaber af et begreb kan defineres i en abstrakt klasse. Definition af detaljerede egenskaber kan finde sted i subklasser, hvor der er mere detaljeret viden til stede.
- Det er muligt at specificere en forudannonceret operation med pre- og postbetingelser.
- Det er ikke muligt at lave instanser af en abstrakt klasse.

Noter

En klasse, som ikke indeholder forudannoncerede operationer, vil vi omtale som en *konkret* klasse.

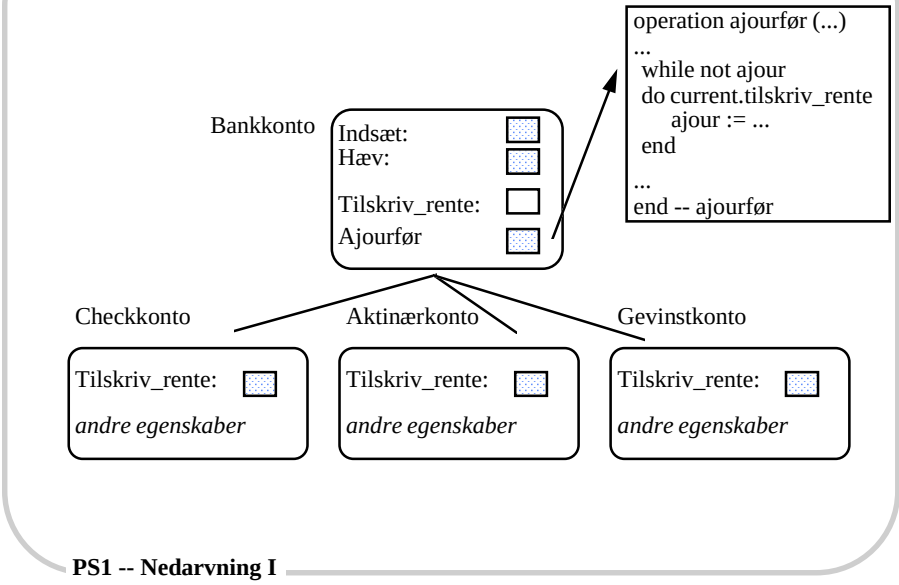
Abstrakte klasser (II).



Noter

Denne slide viser et hierarki af tre klasser; B som arver fra A, og C som arver fra B. A og B er abstrakte klasser. Dog er B "lidt mere konkret" end A, idet operationen Q i B er defineret fuldt ud. C er en konkret klasse, hvori både P og Q er fuldt ud definerede operationer. Endvidere er der kommet en ny operation S til.

Eksempel på anvendelse af abstrakte klasser.



PS1 -- Nedarvning I

Noter

Klassen bankkonto er, som det kan ses, en abstrakt klasse, idet klassen indeholder en operation, som kun er forudannonceret, men ikke defineret fuldt ud.

Denne operation er tilskriv_rente, som ikke kan defineres på det generelle niveau, idet meningen af rentetilskrivningen antages at afhænge af typen af kontoen. Rentetilskrivning til en chekkonto afhænger f.eks. typisk af, hvor mange checks man har brugt i en given periode.

Operationen ajourfør er tiltænkt at skulle føre en konto up-to-date, bl.a. med hensyn til tilskrivning af renter. Hvis det antages, at den eneste måde, ajourføringen afhænger af kontotypen på, er gennem rentetilskrivningen, kan vi programmere ajourfør på det generelle niveau. Gennem dynamisk binding vil ajourfør referere til "den rigtige" rentetilskrivningsteknik.

Ovenstående er meget bemærkelsesværdigt.

Genbrug og nedarvning.

Genbrugelighed af software:

Tilpasning af eksisterende software frem for programmering af nyt.

Klasser, organiseret i et klassehiearki, er velegnede komponenter til både **brug** og **genbrug**:

Brug:
En klasse er *lukket*:

Kan gemmes, oversættes og bruges af klienter.

Genbrug:
En klasse er *åben*:

Kan tilpasses og udvides, uden at forstyrre klienter af den originale klasse.
Kan bruges som superklasse til andre klasser.

PS1 -- Nedarvning I

Noter

Meyer fører en diskussion om disse ting i 10.2.1 i OOSC. Se også afsnit 2.3 i OOSC.

Terminologi.

1. A er superklasse af B (superclass)	Simula	Smalltalk
2. A er basisklasse for B (base class)	C++	
3. A er forgænger til B (ancestor)	Eiffel	
4. A er forældre til B (parent)	Eiffel	

1. B er subklasse af A (subclass)	Simula	Smalltalk
2. B er afledt klasse af A (derived class)	C++	
3. B er efterkommer af A (descendent)	Eiffel	
4. B er arving af A (heir)	Eiffel	

PS1 -- Nedarvning I

Noter

Denne slide viser forskellig terminologi, der benyttes for relationerne mellem en klasse A og en klasse B, som arver fra A.

Vi vil tillade os i disse noter at bruge notationen i flæng og afhængig af hvilken sproglig kultur, vi diskuterer. Vi foretrækker dog den oprindelige Simula/Smalltalk terminologi (nok mere af gammel vane, end fordi det nødvendigvis er den mest klare måde at udtrykke sig på).

Sammenlign den grafiske notation for A og B på denne slide med den tilsvarende på sliden "Udvidelse og specialisering (I)". Notationen på denne slide er naturlig, idet klasserne A og B er syntaktisk adskilte konstruktioner i programteksten. Den grafiske notation fra sliden "Udvidelse og specialisering (I)" giver derimod det bedste billede af objekterne, der instantieres ud fra klasserne A hhv B. Dette er den tilbagevendende skelnen mellem elementerne i en programbeskrivelse (klasse-beskrivelser) og elementer i en programudførelse (objekter).