11
Nedarvning II.

Enkeltnedarvning i Eiffel.

Rename og redefine.

Initialisering af superklasse-dele af et objekt.

Interfaces til klienter og subklasser.

Typesammenlignelighed og polymorfi.

Abstrakte klasser.

Dynamisk binding.

PS1 -- Nedarvning II

Noter

I dette kapitel vil vi studere emnerne fra “Nedarvning I” i en større detalje, og fra et Eiffel perspektiv.

Eiffel klasser med enkelt nedarvning.

Syntaks:

```
class name
 inherit
  superklasse
    rename
     navne tilpasninger
    export
      eksport tilpasninger
    redefine
     redefinitions tilpasninger
    end

creation
  liste af navne på creation procedurer
feature

features

end
```

tilpasning af nedarvede egenskaber fra *superklasse.*

PS1 -- Nedarvning II

Noter

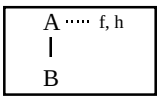
Denne slide viser en skabelon af en klasse i Eiffel i det tilfælde, at der kun arves fra én superklasse.

I det markerede afsnit på billedet ovenfor angives det, hvordan egenskaberne fra superklassen tilpasses, når disse egenskaber skal betragtes som værende (nedarvede) egenskaber i klassen *name.* Bemærk at renaming, export og re-definition skal angives i netop ovenstående rækkefølge.

Der findes to andre tilpasningsmuligheder: **undefine** og **select**. Undefine vil vi ikke bekymre os om i disse noter. (Hvis du er interesseret eller nysgerrig se afsnit 10.16 i "Eiffel the Language"). Select kommer vi ind på i forbindelse med mulitipel nedarving.

I dette kapitel ser vi nærmere på de de forskellige tilpasningsmuligheder.

‘Rename’ i Eiffel.




```
class B
  inherit A
    rename f as g,
           h as j
    andre feature tilpasninger
    end
  ...
feature
  ...
```

Det overordnede problem:

Hvis et navn er defineret i A kan navnet ikke umiddelbart benyttes til en feature i B.

Hvorfor bruge **rename** i klassen B?

- Give adgang til features i superklasser, som har samme navn som i B.
- For at klienter og subklasser af B kan benytte A-features med naturlig terminologi.
- For at løse op for navnesammenfald ved multipel nedarvning.

Noter

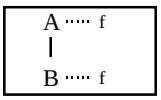
I Eiffel er tilknytningen af et navn til en egenskab (feature) ikke global og absolut, men relativ til en klasse. Gennem **rename**-delen under **inherit** er det muligt at tilpasse navnene på superklasseegenskaber. De nye navne har kun betydning i klassen B (samt dennes subklasser). De gamle navne (navnene på egenskaber i A, som bliver renamet i B’s klassedefinition) er i klassen B totalt løsrevet fra deres A-betydning. De gamle navne kan således bruges som feature-navne i B, hvis vi måtte ønske dette.

Det er noget meget grundlæggende i Eiffel, at navne på features i en klasse entydigt skal referere til netop én feature. I andre sprog, f.eks. C++ kan vi godt bruge det samme navn til en variabel eller operation i en klasse og i dens superklasse, men det er naturligvis også i dette sprog vigtigt, at vi præcist er i stand til at sige, hvilken af variablene eller operationerne vi egentlig ønsker at referere til. I C++ skal man defor i anvendelsessituationen *kvalificere* det tvetydige navn med klasse navnet: class::name.

Den tredie årsag til at bruge **rename** er kun relevant hvis en klasse har mere end én superklasse (multipel nedarvning). Hvis en klasse C arver fra to eller flere klasser, som begge har en feature f, hvilken f skal så gælde i C? En måde at løse dette tvetydighedsproblem på, er at kræve, at mindst én af f-erne 'renames'.

‘Redefine’ i Eiffel.

Kontrolleret overskrivning




```
class B
  inherit A
    evt. renamings
    evt. export tilpasninger
    redefine f
  end
  ...
feature
  ...
  f(...) is
  do
    ...
  end --f
  ...
end --B
```

Ikke arbitrære redefinitioner:

- Signaturen af redefinitionen skal være *sammenlignelig* med den oprindelige signatur.
- Specifikationen af redefinitionen skal være en *delspecifikation* af den oprindelige specifikation.
- En parameterløs funktion kan redefineres til en attribut, men ikke omvendt.

Hvorfor bruge **redefine** i klassen B?

- Tildele en ny mening til navnet i B (Overskrive betydningen fra superklassen).

Noter

Renaming er udtryk for en syntaktisk (næsten kosmetisk) ændring af programmet, eller det kan være nødvendig på grund af 'uheldig' valg af navne i superklasser.

Redefine er derimod udtryk for en mere substantiel, semantisk ændring, hvor meningen af et navn, f.eks. en procedure, ændres. Navnene i under **redefine** er de navne, som er gældende efter evt. tilpasninger efter **rename**.

Det vigtigste at få fat i omkring redefinition er *hvorfor* vi benytter os af det: for at introducere en *ny mening* af en eller andet egenskab, som arves fra superklassen. Men det er naturligvis også væsentligt at forstå de begrænsninger, som redefinition i Eiffel er underlagt.

Én af disse begrænsninger er, at en variabel kan ikke redefineres til en funktion (eller procedure), idet man ikke kan assigne til en funktion.

Signaturen (parameterprofilen) af en feature udtrykker typer af formelle parametre og resultattypen. Man kan også tale om signaturen af en variabel, nemlig som typen af variablen. (Se også forelæsningen om "Specifikation med logiske udtryk"). Se evt. også “Eiffel the Language”, afsnit 5.12 samt 13.3 for en uddybning af signaturbegrebet hhv. signatursammenlignelighed.

Hvad det vil sige, at en specifikation er en delspecifikation af en anden, vender vi tilbage til i næste forelæsning.

Begrænsningerne i de to førte punkter på sliden ovenfor sikrer, at man ikke på må og få kan omdefinere operationer (og variable) i subklasser. Såvel signatur som specifikation, som angivet i en superklasse, skal overholdes af redefinitioner i subklasser.

At signaturen af en redefinition er sammenlignelig med den oprindelige signatur betyder for parametrenes vedkommende, at parametrene parvis er sammenlignelige. Specielt har vi, at en operation f i klassen A og den redefinerede f i B skal have samme antal parametre. En parameter i den redefinerede klasse skal altså være (identisk med, eller) en subklasse af parameteren i den oprindelige klasse. Dette kaldes med et fint ord for *covarians.* Det kan diskuteres, om dette er det rigtige valg. Nogle argumenterer for, at det modsatte (*kontravarians*) skal være gældende. Senere i kurset vender vi tilbage til forskellene mellem covarians og kontravarians.

Klientinterfacet i forbindelse med nedarvning.

- En klasse har fuld kontrol over sin klientinterface relativ til superklassens klientinterface.
- Som grundregel beholder features deres eksportstatus under nedvarning.
- Under nedarvning er det dog muligt at tilpasse nedarvede egenskabers eksportstatus:
  - Tilføje en feature til klientinterfacet.
  - Fjerne en nedarvet feature fra klientinterface.
  - Håndtere alle nedarvede features under ét.
  - Man kan selektivt eksportere til udvalgte klasser (og deres subklasser).

Eksempel 1:

```
class B  -- har features x, y og z
 inherit A
   export
     {ANY} x;
     {C} y;
     {NONE} z
   ...
   end
```

Eksempel 2:

```
class B
 inherit A
   export
      {NONE} all
      {ANY} x
   ...
   end
```

I Eiffel er der ingen form for synlighedkontrol mellem en klasse og dens subklasser.

PS1 -- Nedarvning II

Noter

Generelt overtager en subklasse (B) superklassens (A´s) klientinterface, men superklassen (A) dikterer ikke på nogen måden subklassens (B´s) klientinterface. I export-delen af inheritance af B kan man helt igennem styre B´s interface til sine klienter.

Eksempel 1 ovenfor illustrerer dette. Uanset A´s eksportpolitik vil B, som jo arver x, y og z fra A, stille x tilrådighed for alle klienter, kun eksportere y til C, samt holde z privat.

Eksempel 2 ovenfor illustrerer brugen af nøgleordet **all** i export-delen under inheritance. Meningen er her at *alle* features arvet fra A bliver private (kan ikke ses af nogen klient til A), *undtagen* dog x, som kan ses af alle (ANY) klienter.

Det kan være af interesse at vide, at I Eiffel2 var angivelsen af klientinterfacet væsentlig enklere, men måske lidt mere "træls" at håndtere i praksis. I Eiffel2 havde hver klassen en eksplicit export liste, som hver klasse i et nedarvningshierarki kunne danne helt uafhængig af superklassens export liste. Dog var der mulighed for i en eksportlist at sige "repeat super-klasse", for på denne måde enkelt at kunne udtrykke udvidelse af klient-forhold i forhold til superklassens klient-forhold.

Creation procedurer i forbindelse med nedarvning.

- Hver klasse lister sine creation procedurer. Creation status af procedurer nedarves ikke.
- Gennem evt. anvendelse af renaming er det muligt at benytte superklassens creation procedure (x fra A) i subklassens creation procedure (x i B).
- Creation status af en creation procedure i A (muligheden for at anvende creation proceduren til instantiering) er uafhængig af procedurens eksport status (muligheden for at anvende proceduren på et objekt af typen A).

```
class A
creation x
feature

 x(...) is
 do ... end

end
```

```
class B
  inherit A
  rename x as x_from_A
  export ...
  end
creation x, z
feature

  x(...) is
  do x_from_A(...)  end

  z(...) is
  do ... end

end
```

PS1 -- Nedarvning II

Noter

Denne slide beskæftiger sig med et lignende emne som forrige slide. Spørgsmålet der besvares er hvorledes creation procedurer forholder sig til nedarvning.

Det er væsentligt at slå fast, at udnævnelse af creation procedurer finder sted i hver klasse i et klassehierarki. Man kan ikke blot "læne sig op ad" superklassens creation procedure liste.

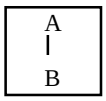
Det er også væsentligt at få fat i hvordan to creation procedurer i to niveauer i et klassehierarki kan bruge hinanden. Altså hvordan B's creation procedure kan kalde A's creation procedure relativ til billedet ovenfor. Dette er et ofte forekommede ønske, idet A´s creation procedure initialiserer A-attributter, og B´s creation procedure initialiserer B´s attriutter. Det vil være dobbeltarbejde hvis vi i B´s creation procedure skulle dublere arbejdet i forhold til A´s creation procedure; det er naturligvis det bedste blot at kunne kalde A´s creation procedure fra B´s. Det er ofte også tilfældet, at creation procedurer hedder det samme på tværs af klasser i et hierarki (typisk make eller create).

Bemærk at vi ikke ovenfor redefinerer x med **redefine**. Hvis vi redefinerede x ville B´s x (bl.a.) være tvunget til at have samme parameterantal som A´s x. Dette kan vi typisk ikke leve med for creation procedurer (der kommer ofte flere og flere parametre på creation procedurer, jo flere superklasser den omkringliggende klasse af proceduren har).

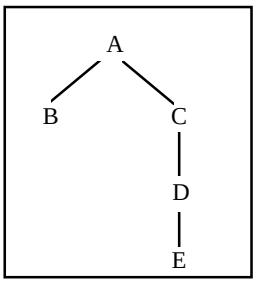
Typesammenlignelighed og polymorfi i Eiffel.

Lad B være en direkte subklasse af A.

B er da *direkte typesammenlignelig* med A.

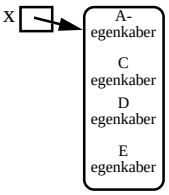


Eksempel:



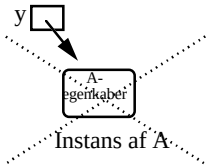
B og C er direkte typesammenlignelige med A.
D og E er typesammenlignelige med A.

x: A;



Instans af E

y: E



Instans af A

PS1 -- Nedarvning II

Noter

Direkte typesammenlignelighed er, som man husker, den primitive relation, som tillader os at definere den mere interessante relation 'typesammenlignelighed'. Sidstnævnte er defineret som den transitive refleksive lukning af 'direkte typesammenlignelighed'.

Bidraget til typesammenlignelighed, som defineret på denne slide, definerer en begrænset og kontrolleret form for polymorfi. En instans af en mere specialiseret klasse kan assignes til en variabel af en mere generel klasse. Dette er en "typesikker" assignment, idet vi ved, at en specialiseret type har alle de egenskaber, som den generelle type besidder.

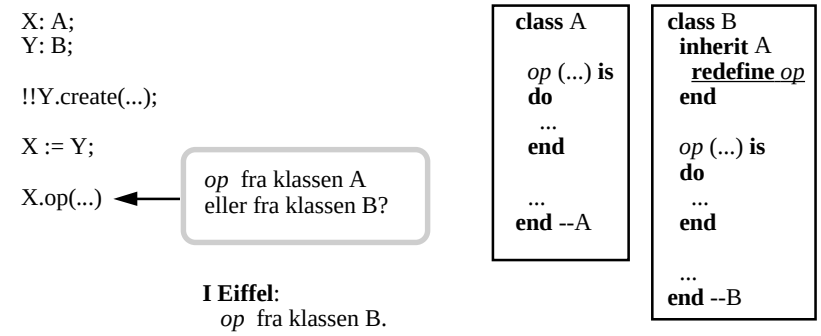
Som det illustreres på denne slide, gælder det omvendte ikke. Hvis det var lovligt at tilskrive et generelt objekt til en mere specifik kvalificeret variabel, kan vi ikke på programmets oversættelsestidspunkt forhindre, at der bliver spurgt om udefinerede egenskaber af objekter.

I eksemplet

y.D-egenskab

går det galt hvis y peger på en instans af klassen A.

Dynamisk binding i Eiffel.



I Eiffel:
op fra klassen B.

Eiffel bruger konsekvent dynamisk binding.x

Betegnelsen 'virtuelle operationer' bruges ikke i Eiffel.

PS1 -- Nedarvning II

Noter

Det kan virke lidt overflødigt skulle skrive '**redefine** op', når det jo klart fremgår af klasse definitionen, at operationen *op* rent faktisk (re)defineres.

Eiffel filosofien omkring dette er, at programmøren skal udtrykke sig lidt redundant (hvilket jo vil sige, at man bruger lidt flere "ord" end nødvendigt), for derved at markere, at operationen ikke blot forekommer ved et uheld både i klassen A og B. Det skal være programmørens intension, at redefinere den udgave af *op*, som han er bevidst om også findes i A.

Det er værd at bemærke, at hvis formålet med

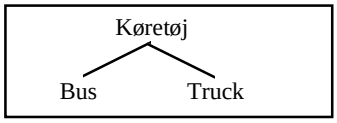
```
!!Y.create(...);
X := Y;
```

udelukkende er at tildele X en reference til et B-objekt kan man nøjes med

```
! B!X.create(...).
```

Dette udnytter den fulde styrke i sprogets objektskabelses mekanisme.

Typesammenlignelighed og reverse assignment attempt.



k: Køretøj;
b: Bus;
t: Truck;

```
!!b.create; !!t.create;

k := b;    -- ok i Eiffel og andre OO sprog.

b := k;    -- ulovligt i Eiffel, men lovligt
           -- i nogle andre OO sprog.

b := t;    -- ulovligt i alle OO sprog.
```

```
!!b.create; !!t.create;

k := b;    -- ok

b ?= k;
if b.void
then ...
end;
```

Eiffels *reverse assignment attempt:* Det testes under programudførelsen om den dynamiske type af k er typesammenlignelig med den statiske type af b.
Hvis ja: udfør normalt assignment.
Hvis nej: assign void til b.

PS1 -- Nedarvning II

Noter

Hovedpointen på denne slide er, at assignmentet b := k kun er lovligt, hvis k refererer til en bus (eller til et objekt, som er en specialiseret bus). Det kan vi kun vide ved at teste det på programmets udførelsestidspunkt. I Eiffel er dette "run time check" en del af det specielle assignment b ?= k, som kaldes et *reverse assignment attempt*.

Bemærk, at et "run time check" aldrig er nødvendig ved assignmentet k := b, idet b altid vil referere til et objekt, som er en bus, eller til et objekt, som er instans af en subklasse af bus.

Et assignment

var := expr

er typemæssigt lovligt hvis typen af expr er typesammenlignelig med typen af var. Et reverse assignment attempt

var ?= expr

er typemæssigt lovligt hvis typen af var er typesammenlignelig med typen af expr; altså præcist den omvendte betingelse. Ovennævnte b ?= k opfylder klart denne regel. Reglen udlukker k ?= b, b ?= t og t ?= b (alle med reference til billedet ovenfor) som lovlige reverse assignment attempts. Reglen er fornuftig derved, at den kun er gyldig, hvis der er tvivl om den typemæssige lovlighed af assignment (forsøget), og hvis der er håb om, at forsøget kan give success.

Man kan læse om reverse assignment attemps i afsnit 20.13 af "Eiffel the Language".

Det er værd at bemærke, at der i klassen ANY findes en funktion som hedder conforms_to, og som på programmets udførelsestidspunkt tillader os at undersøge om et objekts type er sammenlignelig med et andet objekts type. Funktionen bruges således:

x.conforms_to(y).

Svaret er “true” hvis den dynamiske type af x er typesammenlignelig med den dynamiske type af y.

Abstrakte klasser i Eiffel.

Abstrakt klasse med forudannonceret routine f.

```
deferred class  A
  inherit ...
creation create
feature
  ...
  f (...) is
     deferred
  end; --f

  g: T is
     deferred
  end
end
```

```
class B
  inherit A
creation create

feature

   f(...) is
   do
        ...
   end; --f

   g: T
   ...
end
```

```
x: A;
...
!!x.Create
```

```
x: A;  y: B

!!y.Create;
x := y
```

PS1 -- Nedarvning II

Noter

I Eiffel er en klasse abstrakt hvis den indeholder én eller flere features, der er mærkede **deferred**. En abstrakt klasse i Eiffel skal mærkes med nøgleordet **deferred** i forbindelse med klassedefinition.

I eksemplet ovenfor arver klassen B fra klassen A. B siges at *effektuere* (eng.: "effect") feature f og g. Som det ses, er B således ikke abstrakt (vi antager at der ikke er andre abstrakte egenskaber end f og g i A). Vi skal (og må) ikke eksplict skrive

```
redefine f
```

når en egenskab effektueres.

I eksemplet ovenfor ser vi at g er en deferred funktion uden parametre i klassen A. I klassen B kan vi naturligvis effektuere g som en funktion. Men som illustreret, er det også muligt at effektuere g som en attribut. Man kan således se, at formen

```
g: T is
  deferred
end
```

i A lader det stå åbent, om vi i en subklasse vil realisere g som en beregning af en værdi (en parameterløs funktion) eller som en lagret værdi (en attribut).

Undertiden omtaler man en klasse som en "effective class" (i betydningen at den er af egentlig nytte, tjenestdygtig) hvis den ikke er deferred.

Abstrakt klasse med specifikation: Stack (I).

```
class STACK[T]
feature
  nb_elements: INTEGER is deferred
  end;

  push(x: T) is
  require not full
  deferred
  ensure not empty;  top = x;  nb_elements = old nb_elements + 1;

  pop is
  require not empty
  derferred
  ensure not full;  nb_elements = old nb_elements -1
  end; --pop

  top: T is
  require not empty
  deferred
  ensure Nochange
  end; -- top
--fortsættes
```

PS1 -- Nedarvning II

Noter

Vi benytter pøny stakke som et eksempel.

Der er ingen repræsentationsaspekter i Stack. Repræsentationsbeslutninger udsættes til en subklasse.

nb_elements er nu ikke en integer, men en forudannonceret feature af typen integer, hvis implementation afhænger af repræsentationsvalget. Vi kan således i en subklasse vælge at lagre eller beregne antallet af elementer.

Vi ønsker ikke apriori at begrænse stakke til at have en øvre grænse. Dette afspejles af invarianten derved, at der ikke er angivet en øvre grænse af nb_elements. Vi kunne vælge at realisere stakke med en dynamisk datastruktur (linærere lister), som ikke har en øvre grænse på antallet af elementer. I dette tilfælde skal vi i en subklasse til stack implementere full som en funktion, der altid returnerer false.

Abstrakt klasse med specifikation: Stack (II).
(fortsat)

```
    empty: BOOLEAN is
    require true
    do Result := (nb_elements = 0)
    ensure Result = (nb_elements = 0)
    end;

    full: BOOLEAN is deferred
    end;--full

    change_top(x: T) is
    require not empty
    do pop; push(x)
    ensure not empty; top = x; nb_elements = old nb_elements
    end; --change_top

    invariant
      0 <= nb_elements;

  end --class stack
```

PS1 -- Nedarvning II

Noter

Change_top er en ny operation. Operationen udskifter toppen af stakken med parameteren. Denne operation er ikke forudannonceret, idet vi, *uanset* specialiseringen af stack, kan implementere change_top på dette niveau.

Eksempel på en abstrakte klasse fra Eiffel bibliotekerne.

```
deferred class PART_COMPAR
feature
    infix "<" (other: like Current): BOOLEAN is
        deferred
        end; -- "<"

    infix "<=" (other: like Current): BOOLEAN is
        do
            Result := (Current < other) or (is_equal (other))
        end; -- "<="

    infix ">" (other: like Current): BOOLEAN is
        do
            Result := other < Current
        end; -- ">"

    infix ">=" (other: like Current): BOOLEAN is
        do
            Result := (other < Current) or (is_equal (other))
        end -- ">="

end -- class PART_COMPAR
```

PS1 -- Nedarvning II

Noter

Klassen PART_COMPAR definerer fire forskellige sammenligningsoperationer på objekter, hvorpå vi introducerer en *partiel ordning* (<). Vi minder om, at en partiel ordning pr. definition er

transitiv (x < y og y < z => x < z) og

irrefleksiv (x < x er falsk for alle x).

Via transitivitet og irrefleksivitet har vi også

anti-symmetri (x < y => not (y < x)).

Klassen PART_COMPAR er fra kerne-biblioteket i vores Eiffel system.

Det ses hvordan operatorerne <=, > og >= kan defineres ud fra < og equal. Bemærk at < er den eneste forudannoncerede (deferred) operation i klassen. Is_equal er arvet fra klassen ANY, hvorfra enhver klasse i Eiffel arver. Det er nu intentionen, at subklasser af PART_COMPAR skal definere operationen <; derved får man de andre "gratis".

Eksemplet viser også brug af såkaldte forankrede typer i Eiffel (type erklæringer som 'other: **like** Current') og infix operator funktioner ('**infix** "<"(...): boolean'). Hverken forankrede typer eller infix operator funktioner bliver behandlede eksplicit på dette kursus. Interesserede henvises til hhv. afsnit 12.15 "Eiffel the Language" angående forankrede typer.