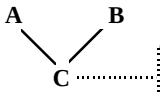
12
Nedarvning III.

Multipel nedarvning.

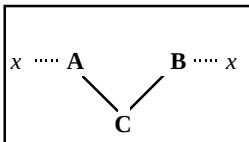
Nedarvning og assertions.

Noter

Multipel nedarvning.



- Klassen C arver egenskaberne fra både A og B.
- A og B egenskaber er umiddelbart tilgængelige i C.



x er en egenskab i både klassen A og B.
c er en instans af klassen C.

c.*x* :

x fra A eller
x fra B?

- **Navnesammenfaldsproblemet**: Hvilken egenskab refererer *x* til i C?
- **Kombinationsproblemet**: Kan *x* fra A og B kombineres i C?
- **Selektionsproblem**: Kan man fra C vælge mellem x fra A eller B?
- **Replikationsproblemet**: Ønsker vi to *x*-er i C?

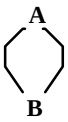
PS1 -- Nedarvning III

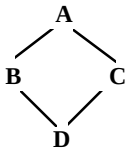
Noter

Når vi taler om navnesammenfald som et problem er det ud fra en filosofi om, at et navn kun kan referere til én egenskab (én routine og/eller én attribut). Hvis flere engenskaber har samme navn er det tvetydigt, hvilken egenskab vi egentlig mener.

En alternativ filosofi består i, at vi refererer til begge (alle) egenskaber i "en eller anden kombination". Kunststykket er naturligvis så at finde ud af, hvordan man generelt, eller for bestemte slags egenskaber, kan danne kombinationer. Vi vil senere i denne forelæsning vende tilbage til kombinationsproblemet.

Gentagen nedarvning.





Direkte gentagen nedarvning

Man arver to eller flere gange fra den samme klasse.

Forekommer i Eiffel hvis man ønsker at replikere en attribut i A.

Forekommer også hvis man ønsker både at redefinere og rename en feature fra A.

Indirekte gentagen nedarvning

Forekommer meget ofte i Eiffel, uden at man er bevidst om det, eksempelvis med A som ANY.

Man ønsker meget sjældent at replikere egenskaberne i A

En klasse kan ikke arve direkte eller indirekte fra sig selv.

PS1 -- Nedarvning III

Noter

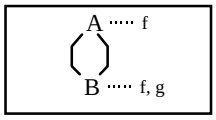
Gentagen nedarvning kaldes i den engelske Eiffel litteratur for *repeated inheritance.*

På den næste slide vil vi set på et praktisk forekommende eksempel på direkte gentagen nedarvning i Eiffel.

Normalt anvender vi ikke pile når vi illustrerer nedarvning grafisk. Der er en implicit orientering, således at superklasser altid er øverst, og subklasserne er nedenunder.

Nederst på denne slide illustrerer vi, at nedarvning fra sig selv er forbudt. Så der er altså grænser...

‘Rename’ og ‘redefine’ idiom i Eiffel.



Situation:

- Operationen f er defineret i klassen A.
- Operationen f redefineres i B.
- B's operation f skal kalde A's operation f.

- g er det nye navn for f fra A.
- f i B redefinerer f fra A.
- f i B kalder gennem navnet g f fra A.

```
class B
inherit A
  rename f as g end
inherit A
  redefine f
  select f end
feature

  f(...) is
  do
    ...
    g(...)
    ...
  end --f
end --B
```

PS1 -- Nedarvning III

Noter

Denne slide viser en sproglig udtryksform, der er så typisk, at vi kalder det et idiom.

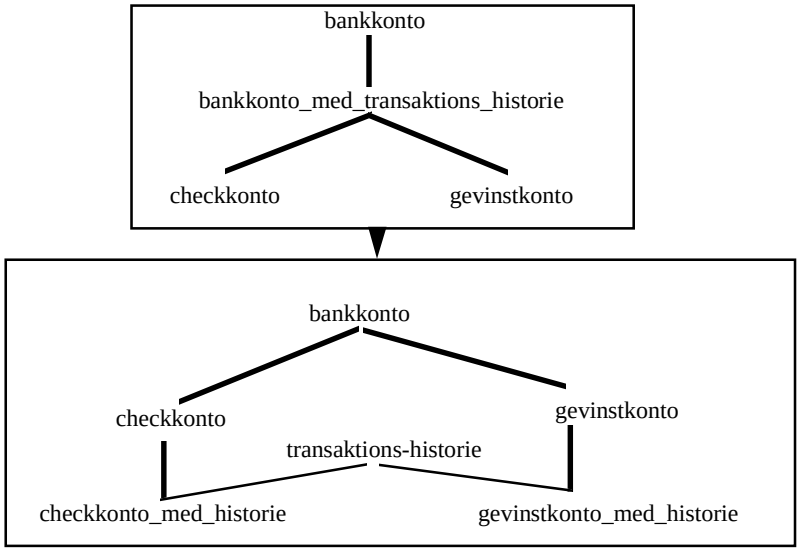
Et idiom er en (ikke teknisk) betegnelse for en typisk sproglig udtryksform,

Det beskrevne idiom er et, man efterhånden tilegner sig, uden nødvendigvis ved hver anvendelse at tænke dybt over, hvad man egentlig udtrykker.

Gennem gentagen nedarvning fra A kan vi både rename og redefinere operationen f. Derved kan den redefinerede f i B kalde g, som jo altså er den feature, vi i A kalder f.

Select er en mekanisme, der i Eiffel kun anvendes i forbindelse med gentagen nedarvning. Den fortæller generelt hvilken version af f der skal være gældende, i tilfælde af dynamisk binding. Interesserede læsere henvises til 11.6 og følgende afsnit i “Eiffel the Language”.

Eksempel på anvendelse af multipel nedarvning (1).



PS1 -- Nedarvning III

Noter

Denne slide viser igen et klassehierarki for forskellige bankkonti. Klassen bankkonto_med_transaktions_historie udvider klassen bankkonto med en hukommelse, hvor hver enkelt transaktion på kontoen registreres. Dette kan f.eks. senere bruges som basis for kontoudtog.

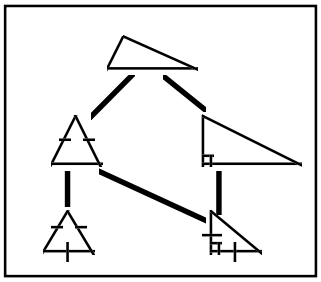
På den nederste del af illustrationen er vist, hvordan transaktionshistorien er faktoriseret ud og mixet sammen med de to forskellige kontotyper ved brug af multipel nedarvning.

Det giver formodentlig ikke mening at instantiere transaktions_historie uden en eller anden bankkonto.

På denne og de næste slides er de super-subklasse relationer, der repræsenterer specialisering tegnet med fede linier. Andre anvendelser af super-subklasse relationen er tegnet med tynde linier.

Eksempel på anvendelse af multipel nedarvning (2).

Et hierarki af trekantstyper.



PS1 -- Nedarvning III

Noter

I hierarkiet af trekanter arver en ligebenet retvinklet trekant både fra ligebenede trekanter og fra retvinklede trekanter.

En ligebenet retvinklet trekant *er* både en ligebenet trekant og en retvinklet trekant.

Bemærk forskellen mellem trekanteksemplet og bankkontoeksemplet fra forrige slide. Vi kan ikke på samme måde sige, at en checkkonto med historie *er* en checkkonto og en transaktionshistorie.

Eksempel på anvendelse af multipel nedarvning (3).

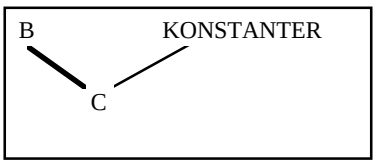
Problem:

Der findes ikke et oplagt globalt sted i et Eiffel program, hvor delte definitioner kan beskrives.

Løsning:

Konstanter og delte objekter (i termer af "once funktioner") defineres i en klasse, f.eks. i klassen KONSTANTER.

Alle klasser, som har brug for adgang til konstanterne og de delte objekter, *arver* fra KONSTANTER.



To anvendelser af multipel nedarvning:

C er en specialisering af B ("C *er* en B").

KONSTANTER er blot *gjort tilgængelig* for C

PS1 -- Nedarvning III

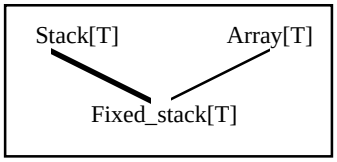
Noter

Vi har i kapitlet "Basal objekt-orienteret programmering II" diskuteret, hvordan man kan definere konstanter i Eiffel. I denne sammenhæng introducerede vi de såkaldte once-funktioner, som også omtales på denne slide.

Sammenlign problemet, som rejses på denne slide, med det tilsvarende problem, og ikke mindst den oplagte løsning, i Pascal og de fleste andre sprog. Løsningen er naturligvis at placere definitionerne, som vi ønsker at dele, på et globalt tilgængeligt sted i programmet. I Pascal er dette sted på det yderste scopeniveau; de fleste sprog har et sådant “globalt sted”. Men Eiffel har ikke. Filosofien i Eiffel er den, at mange definitioner et sådant sted kan sænke program-genbrugsværdien. Man kan ikke løsrive dele af programmet, og flytte disse til et andet sted uden også at flytte de delte programdefinitioner (fra det globale sted) med.

Eksempel på anvendelse af multipel nedarvning (4).

Fornuftsægteskab.



- Stack er en abstrakt klasse, som ikke indeholder repræsentationsbeslutninger.
- Fixed_stack er en konkret klasse
  - Arver den noble funktionaliet fra Stack.
  - Arver den substantielle funktionaliet fra Array.

Noter

Dette fornuftsægteskab mellem på den ene side

> den fornemme Stack, som har en række fine egenskaber på facaden, men er tom inden i

og på den anden side

> det solide Array, som har en række primitive egenskaber uden at vide, hvordan de kan anvendes på et højere plan

er beskrevet nærmere i OOSC afsnit 10.4.1.

Anvendelser af multipel nedarvning.

Opsamling.

- **"Begrebsmæssig sund multipel nedarvning"**.
  Specialisering af to klasser, som har overlappende ekstensioner.
- **Umiddelbar tilgang til nogle definitioner.**
  Når man har behov for at stille en samling af definitioner til rådighed for en klasse, f.eks. konstanter eller et matematisk funktionsbibliotek.
- **Uniform aggregering**.
  Uniform aggregering af egenskaber fra to eller flere klasser.
- **"The mariage of convenience".**
  I forbindelse med konkretisering af abstrakt klasse, hvor egenskaber fra en anden klasse er nødvendige.

PS1 -- Nedarvning III

Noter

Denne slide opsamler nogle årsager til at benytte sig af multipel nedarvning. Der er vist eksempler på de fleste af disse ting på tidligere slides i dette kapitel.

Hierarkiet af trekanter giver et eksempel på “begrebsmæssig sund multipel nedarvning” fordi de bagvedliggende begreber er naturlige specialiseringer af hinanden.

Uniform aggregering er eksemplificeret gennem bankkonto eksemplet. Transaktions-historie og f.eks. checkkonto egenskaber er slået sammen, og "summen af egenskaber" er gjort tilgængelig i klassen checkonto_med_historie. Denne aggregering er uniform, idet man ikke i checkonto_med_historie kan se, hvorfra den enkelte egenskab stammer.

Der er måske kun en nuanceforskel mellem “uniform aggregering” og “umiddelbar tilgang”. Vi tænker dog på “umiddelbar tilgang” som en måde at stille definitioner (såsom funktioner og konstanter) tilrådighed, hvorimod uniform aggregering adderer substans (typisk attributter og tilhørende naturlige operationer).

Teknikker til løsning af navnesammenfalds-problemet.

1. *På navne-anvendelsestidspunktet:*
   Krav om eksplicit programmør-angivelse af det foretrukne navn.
2. *På klasse-definitionstidspunktet:*
   Ændring af navne, som er i konflikt med hinanden, når der dannes en klasse med multiple superklasser.
3. *På klassedefinitions-tidspunktet:*
   Fastlæggelse af dominans-rækkefølge af klasser.
4. Tilfældig (non-deterministisk) valg af navn, som indgår i en navnekonflikt.
5. Forbud mod navnesammenfald.

PS1 -- Nedarvning III

Noter

På denne slide opremser vi en række teknikker til løsning af navnesammenfaldsproblemet.

Den første løsning er anvendt i C++. Den består i, at hver anvendelse af et tvetydigt navn skal "påklistres" (kvalificeres med) information, som gør det kvalificerede navn utvetydigt. I C++ skriver man klasse_navn::navn.

Den anden løsning er Eiffel løsningen, ved brug af rename.

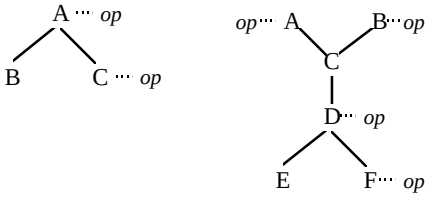
Den tredie løsning kendes fra bl.a. CLOS, the Common Lisp Object System, hvor rækkefølgen af superklasserne i en klasse bliver afgørende for, hvilke egenskaber der kommer til at dominere.

De to sidste løsninger kendes vist nok ikke fra noget (objekt-orienteret) programmeringssprog.

Kombinationsproblemet.

Hvilke muligheder for kombination af egenskaber har man i tilfælde af navnesammenfald?

- Kombination af operationer (*metodekombination*) som har samme navn.
- Giver mening ved både enkelt og multipel nedarvning.



Imperativ metodekombination

- Eksplicit programmering af et samarbejde mellem *op* operationerne.

Deklarativ metodekombination

- Baseret på en abstraktion over et fast samarbejdsmønster.

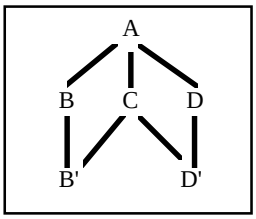
PS1 -- Nedarvning III

Noter

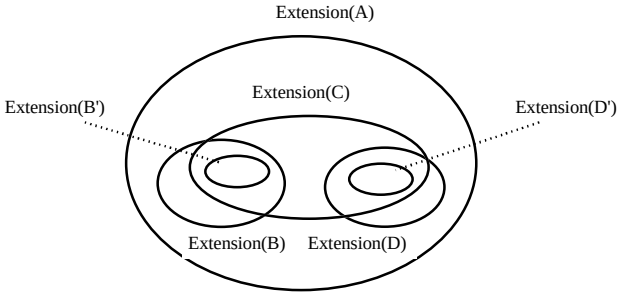
Kombination af operationer med samme navn i et nedarvningshierarki er et vigtigt emne i mange objekt-orienterede programmeringssprog. I tilfældet til venstre ovenfor (hvor C arver fra A) er det almindeligt at *op* i C påkalder sig *op* i A. Nogle sprog har specielle sproglige mekanismer der gør det enkelt at programmere dette (imperativ metodekombination). Det varierer fra sprog til sprog om det er *op* i C eller *op* i A som har kontrollen over kombinationen. I Eiffel bruger man “rename-redefine” idiomet (se tidligere i dette kapitel) til a berede vejen for en kombination af *op* fra A og C.

Deklarativ metodekombination er først og fremmest kendt fra objekt-orienterede sprog i Lisp familien: Flavors og CLOS. Deklarativ metodekombination kan basere sig på, at alle metoder, som har samme navn, har tilknyttet en *rolle*. I CLOS har man bl.a. en before-rolle, en primær rolle og en after rolle. Én af de deklarative metodekombinationer forløber ved, at alle before-metoder kaldes, hvorefter den mest specifikke primære metode kaldes, hvorefter alle after-metoder kaldes. Alt dette er meget spændende! Interesserede læsere henvises til CLOS litteraturen for yderligere detaljer (f.eks. Keene: Object-Oriented Programming in Common Lisp, Addison Wesley).

“Begrebsmæssig sund” multipel nedarvning.



- C-objekter er af typen A.
- Der er et overlap mellem ekstensionen af typen B og C (ditto mellem D og C).
- Både B og D klasser med og uden C egenskaber er tilstede.



PS1 -- Nedarvning III

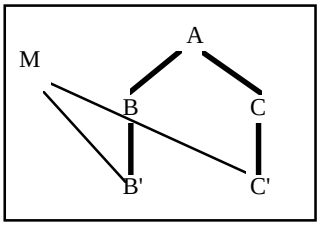
Noter

Denne slide tager udgangspunkt i, at hvis B er en subklasse af A, da er ekstension(B) en delmængde af ekstension(A). Dette har vi diskuteret allerede i den første forelæsning om nedarvning.

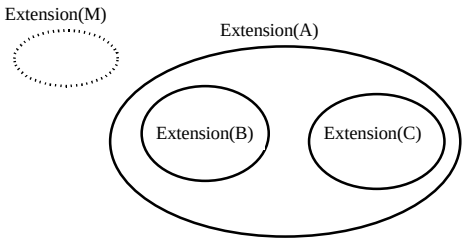
For begrebsmæssig sund multipel nedarvning vil vi kræve, at dette krav om indlejring af ekstensioner holder for alle de specialiserede klasser. Dette giver billedet som vist ovenfor.

Eksemplet hvor multipel nedarvning anvendes i forbindelse med definition af et klassehierarki for trekanter repræsenterer, hvad vi kalder begrebsmæssig sund multipel nedarvning.

Multipel nedarvning med 'mixins'.



- Egenskaberne i M virker som et *krydderi* på egenskaberne i hhv. B og C.
- Både de krydrede og ikke-krydrede klasser er tilgængelige.



PS1 -- Nedarvning III

Noter

I dette tilfælde er der ikke noget overlap mellem ekstensionerne af A og M.

Det vil typisk være tilfældet, at der overhovedet ikke eksisterer instanser (objekter) af typen M. M kan kun meningsfuldt instantieres når det sker "sammen med andre klasser". Derfor har vi tegnet extension(M) med stiplede linier.

Det vil endvidere typisk være sådan, at klasserne B' og C' er helt trivielle: De indeholder ikke andet end en inherit B, M (hhv inherit C, M) samt evt. renamings. Visse steder i litteraturen kaldes sådanne klasser 'aggregate classes', fordi de udelukkende samler egenskaber fra to grene. Aggregeringen er dog speciel derved, at egenskaberne sidestilles; vi kan ikke se forskel på, om en egenskab kommer fra den ene eller den anden gren. Dette har vi tidligere omtalt som uniform aggregering.

Multipel og/eller gentagen nedarvning i Eiffel.

```
class name  inherit
  superclass-1
     rename renamings
     export export adaptations
     redefine redefinitions
     end;
  ...
  superclass-n
     rename renamings
     export export adaptations
     redefine redefinitions
     end

creation list of creation proc
feature

  features

end
```

Feature tilpasning

- For hver superklasse er der mulighed for "feature tilpasning".
- Ved navnesammenfald blandt superklassernes features kan man via renaming opnå
  - sammenfald, hvis navnene på egenskaberne er ens (efter evt. renaming)
  - replikation, hvis navnene er forskellige.

Noter

Ovenfor har vi vist en forholdsvis generel skabelon for en klasse der arver fra *n* superklasser. Der er mulighed for featuretilpasning for hver superklasse. Vi har nævnt rename, redefine og export tilpasningerne. Der findes to andre, som der dog er betydelig mindre behov for. Dette er select (som vi har set anvendt i forbindelse med rename-redefine idiomet tidligere i denne forelæsning). Det andet er undefine, hvormed man kan gøre en effektiv feature abstrakt (deferred). Se kapitel 10 i "Eiffel the Language", som handler om feature tilpasning.

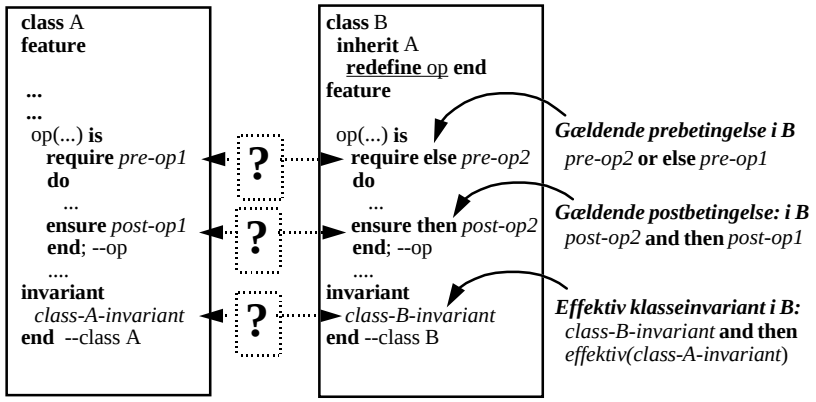
Hvis man ikke har behov for featuretilpasning kan man helt springe det over. Derved får vi følgende mere simplere skabelon:

```
class name  inherit
  superclass-1;
  superclass-2;
  ...
  superclass-n

creation list of creation proc
feature

  features

end
```

Nedarvning og 'assertions' (I).

Problemet:

Hvordan virker specifikationen af en subklasse i forhold til specifikationen af superklassen?



PS1 -- Nedarvning III

Noter

Pre- og postbetingelserne samt klasseinvarianten udgør en specifikation af den datatype, som klassen definerer.

Denne slide beskriver hvordan *gældende prebetingelse* og *gældende postbetingelse* af eksporterede operationer dannes på tværs af nedarvning.

Lad mig her minde om, at vi også har talt om effektive pre- og postbetingelser, nemlig som konjunktionen (**and**-ing) af de formulerede betingelser og klasseinvarianten. Dette var før nedarvning kom ind i billedet. Vi definerer nu den effektive prebetingelse af en operation op i en klasse B som

> den gældende prebetingelse af op i B **and then**
>
> den effektive klasseinvariant af B.

Vi har en ganske tilsvarende definition for postbetingelsen.

Hvis en prebetingelse ikke angives eksplict i en redefineret operation (såsom i *op* i B), opfattes den som værende **false**. Dette er hensigtsmæssigt idet

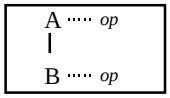
> false **or else** pre-op-1 = pre-op-1

Tilsvarende, hvis en postbetingelse ikke angives eksplicit. I denne situation opfattes postbetingelsen som værende **true**.

Sliden definerer også den effektive invariant af klassen B, som konjunktionen af den formulerede invariant i B og (rekursivt) den effektive invariant af A. I det viste tilfælde (hvor A ikke arver fra nogen klasse - i det mindste eksplict) kan vi betragte class-A-invariant som værende lig med den effektive class-A-invariant.

Assertions i forhold til redefinition beskrives i afsnit 10.15 af “Eiffel the Language”, hvortil der henvises for yderligere detaljer.

Nedarvning og 'assertions' (II).



Prebetingelser:

Prebetingelsen af *op* i B er svagere
end den tilsvarende prebetingelse af *op* i A.

Postbetingelser:

Postbetingelsen af *op* i B er stærkere
end den tilsvarende postbetingelse af *op* i A.

En operation *op* i en specialiseret klasse

- må ikke forhindre at arbejdet bliver udført
- skal gøre arbejdet mindst lige så godt som en operation *op* i en mere generel klasse.

Invarianter:

Den effektive klasseinvariant i B er stærkere end
den effektive klasseinvariant i A.

PS1 -- Nedarvning III

Noter

Denne slide skal forstås som en forlængelse af den forrige slide. Operationen *op* redefineres altså i klassen B i forhold til op i klassen A.

På denne slide udtrykkes det, hvad det vil sige at specifikationen af en operation er en *delspecifikation* af en anden.

Hvad betyder det at "A1 er stærkere end A2" når A1 og A2 er boolske udtryk? Vi definerer det til at betyde at A1 => A2.

Tilsvarende betyder "A1 er svagere end A2" at "A2 er stærkere end A1", altså at A2 => A1.

Der mindes om, at de pre- og postbetingelser, vi skriver i et Eiffel-program kaldes de formulerede pre- og postbetingelser. På forrige slide diskuterede vi de gældende prebetingelser og gældende postbetingelser, der tog pre- og postbetingelser i superklasser i betragtning. De gældende pre- og postbetingelser skal konjungeres med de effektive klasseinvarianter for at få de egentlige betingelser, som skal gælde før og efter operationsaktivering.

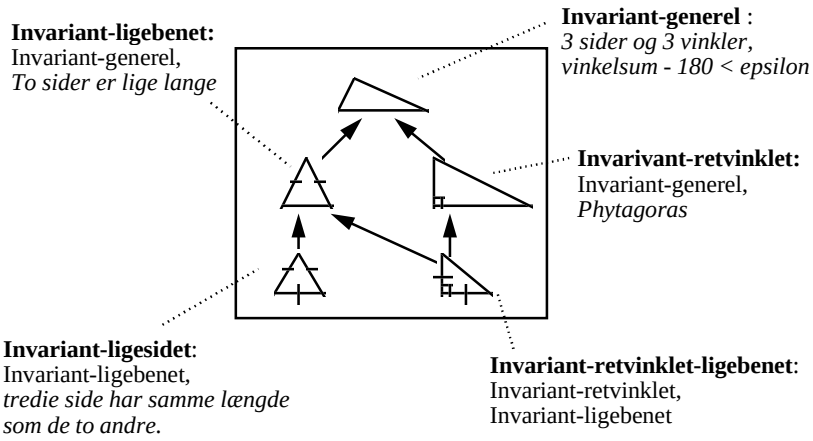
Hvad angår klasseinvarianten skal vi kun angive styrkelsen af denne i subklassen B. Med andre ord er den effektive invariant af B lig med

den effektive invariant af A **and** den formulerede invariant af B.

Herved bliver det automatisk således, at invarianten altid styrkes fra en klasse til en subklasse af denne.

Man kan sige, at i Eiffel3 nedarves både pre- og postbetingelser og klasseinvarianten. De nedarvede assertions kombineres på forskellig vis med de formulerede assertions. I Eiffel2 var det kun klasseinvarianten, der blev nedarvet.

Eksempel på nedarvning og invarianter.



PS1 -- Nedarvning III

Noter

Hver invariant er navngivet på denne slide. Navnet er skrevet med fede typer.

Den effektive invariant er angivet for hver trekantstype. Kursiverede udsagn er de udsagn, som knytter sig direkte til de enkelte typer af trekanter. De kursiverede udsagn er altså de formulerede invarianter af klasserne.

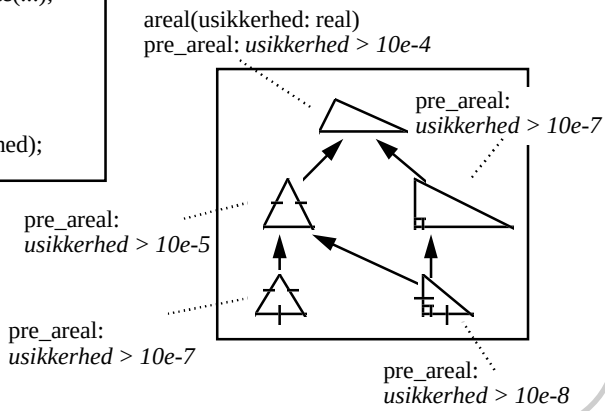
Bemærk, at retvinklede, ligebenede trekanter ikke bidrager med egne udsagn. Disse trekanters invariant er konjunktionen af de to bidrag fra superklasserne.

På næste slide har vi igen hierarkiet af trekanttyper samt operationen, der beregner arealet af de forskellige typer. Antag, at areal-operationen er defineret på alle trekantstyper. Til højre på næste slide er pre-betingelserne af disse areal-operationer givet. Areal-operationen har en parameter, der angiver den mindste usikkerhed, hvormed areal operationen kan kaldes. I takt med, at vi specialiserer trekantstyperne kan vi beregne arealet mere og mere eksakt. Dette betyder, at vi tillader mindre usikkerheder for mere specialiserede trekantstyper. Dette kan ses ved at sammenligne prebetingelserne. Bemærk at udsagnet "usikkerhed > 10e-7" er svagere end f.eks. udsagnet "usikkerhed > 10e-5".

(Fortsættes)

Eksempel på nedarvning og pre-betingelser.

```
t: generel_trekant;
t1: ligebenet_trekant;
t2: retvinklet_ligebenet_trekant;

t1.create(...);    t2.create(...);

if some_condition
  then t := t1
  else t:= t2
end;
...t.areal(max_usikkerhed);
```



PS1 -- Nedarvning III

Noter

(Noter er startet nederst på forrige noteark)

Usikkerhedsparameteren bliver anvendt i areal-operationernes postbetingelse på f.eks. denne måde: abs(result - *rigtig-aral*) < usikkerhed. I dette eksempel vil det nok være svært at udtrykke *rigtig-areal*.

Bemærk at ovenstående kursiverede udsagn (såsom *usikkerhed > 10e-7* for retvinklede trekanter) er den formulerede prebetingelse:

> **require else** usikkerhed > 10e-7 .

Disjunktion (or-ing) med prebetingelsen af generelle trekanter efterlader denne formulerede prebetingelse uændret.

I tilfældet retvinklet, ligebenet trekant, som arver fra ligebenet hhv. retvinklet trekant bliver den gældende prebetingelse

> (usikkerhed > 10e-5) **or else** (usikkerhed > 10e-7) **or else** (usikkerhed > 10e-8)

som jo koger ned til usikkerhed > 10e-8.

En variabel t, som er af den mest generelle trekantstype, kan tilskrives trekanter af mere specifikke trekantsobjekter. Dette er polymorfi. Hvis arealoperationen anvendes på t vil den mest specifikkeareal operation blive anvendt i forhold til den dynamiske type af t. Dette er dynamisk binding.

Filosofien er nu den, at prebetingelsen af operationer i specifikke klasser ikke må stille hindringer i vejen for, at de bliver kaldt. Ved at se på programmet (statisk betragtning) kan vi ikke vide om t er en ligebenet trekant eller en retvinklet ligebenet trekant.

Meyer ønsker at anlægge den mest defensive betragtning: Selv om en meget specifik arealoperation skulle blive kaldt, må dens prebetingelse ikke være stærkere end generelle operationers prebetingelser. Med andre ord: specifikke operationer skal altid kunne kaldes, hvis deres mere generelle modstykker kan kaldes.

Postbetingelsen af specifikke operationer kræves tilsvarende at være stærkere end postbetingelserne af de tilsvarende mere generelle operationer. Dette er nok et mere intuitivt rimeligt krav, og det svarer helt til stramningen af invarianten. Så det gør vi ikke mere ud af her.