29
Opsamling af Objekt-orienteret Programmering.

Bottom-up kontra top-down design.

"The shopping list approach".

Hvordan finder man *på* objekterne.

Klasser og dataabstraktion.

Klasse interface og interface-teknikker.

Klasse nedarvning og nedarvnings-teknikker.

'Køber' eller 'arving'?

Co-varians og contra-varians.

Indskrænkning af klient-interface.

PS1 -- Opsamling af OOP

Noter

Bottom-up kontra top-down design.

Bottom-up design:

abstraktioner over definitioner, som indkapsler data (= klasser).

Små enheder, som er *data-abstraktioner*, **tilpasses** og **kombineres** til gradvis større enheder.

Enhedernes genbrugsværdi er relativ stor.

Top-down design:

abstraktioner over kommandoer (= procedurer) eller udtryk (= funktioner).

Store enheder, som er *procedure-abstraktioner*, **forfines** og **nedbrydes** til gradvis mindre enheder.

Enhedernes genbrugsværdi er relativ lille.

Objekt-orienteret programmering lægger primært op til bottom-up design.

PS1 -- Opsamling af OOP

Noter

En klasse tilbyder en samling af tjenester.

En klasse C tilbyder en samling af tjenester (= features) til kunder (= klienter).

- Tjenester fra en klasse er ligeværdige.
- Tjenester forekommer separat, og løsrevet fra bestemte anvendelsesordener og -kombinationer.
- Mængden af tjenester af passende afrundet.

Hvormange tjenester må der være i en klasse?

- Tilgodese kravene om separation og afrundethed af tjenester. **+**
- Tilgodese overskuelighed, forståelighed og læsbarhed. **-**
- Faktoriser variationer og basale egenskaber ud i hhv. sub- og superklasser. **-**

En klasser *udfører* ikke noget. Den *tilbyder* noget.

Noter

Ligeværdighed: Det er udelukkende varens værdi for kunden, der afgør, om den bliver købt.

Minimale bånd på kombination og orden: Det er kunden som bestemmer i hvilken kombination og orden egenskaberne skal anvendes. Selvom to operationer i en given anvendelsessituation altid bruges sammen, er der gode grunde til ikke at svejse dem sammen i klassen. Selv om nogle mennesker altid spiser sovs til kartofler, er det hensigtsmæssigt at ingredienserne sælges hver for sig.

Afrundethed: for at opnå den bedste genbrugsværdi, uden snæver hensyntagen til en specifik brugssituation.

På ovenstående figur betyder et '+' en forøgelse af antallet af tjenester, og et '-' en formindskelse.

Hvordan finder man *på* klasser og objekter

Inspirationskilder:

- ***Begreber og fænomener i systemets omverden.***

  Begreber bliver til klasser.
  Fænomener bliver til objekter, som er instanser af klasser.

- ***Klasser i programmeringsomgivelsens bibliotek.***

  Tilpasning af eksisterende klasser
  er mere attraktiv end
  Opfindelse af nye klasser.

- ***Systematisk dataoverførsel mellem en mængde af relaterede procedurer eller funktioner:***

  Indfør en dataabstraktion (en klasse),
  hvori procedurerne og funktionerne er operationer.

Noter

Det sidste tilfælde (procedurer med systematisk dataflow) er beskrevet i stor detalje, og ud fra et eksempel, i 12.3 af OOSC.

Gode klasser er dataabstraktioner.

- En klasse skal beskrive en mængde af objekter.
  - Typisk flere objekter af samme klasse.
  - Det skal være naturligt at lave instanser af klassen.
- Objekter skal have tilstand, som på ethvert tidspunkt i dets levetid skal være meningsfuld.
  - Meningsfuldhed: udtrykkes af invarianten.
- Der skal være relevante og meningsfulde operationer på objekterne.
  - Relevans: operationerne skal "arbejde på" objekternes tilstand.
  - Meningsfuldhed: Operationer skal respektere invarianten.

Noter

Klasser beskriver og dækker over objekter: Antag man har en klasse, og man ønsker at benytte en operation i klassen. Antag endvidere at dette dybest set burde kunne lade sig gøre, uden at lave et objekt af klassen. Det objekt-orienterede programmeringssystem tvinger dog programmøren til at lave en dummy instans af klassen, således man kan sige dummy.nyttig-operation(parametre). **Klassen er i dette tilfælde ikke en dataabstraktion.**

Man kan forestille sig en klasse med en operation, som blot benytter det omkringliggende objekts tilstand til intern bogholderi information. Eksempel: Operation som genererer tilfældige tal. Operation med hukommelse. **En sådan klasse er ikke en dataabstraktion.**

Eksempel på en dårlig klasse.

```
Class  udskriv_regning_operation

feature {NONE}
  Kunde_navn: string;
  Vare_liste: list[varer];

  hent_regnings_info(info_kilde: XX) is
  do
    definer Kunde_navn og Vare_liste
  end;

feature

  udskriv_regning(info_kilde: XX;
                  info_destination: YY) is
  do
     hent_regnings_info(info_kilde);
     udskriv regningen på info_destination
  end

end
```

Klassen er entydigt bygget op omkring en handling, ikke om data.

Klassen beskriver ikke tilstand, som er meningsfuld i den fulde livstid af instanser.

Det er unaturligt at instantiere klassen.

Operationerne i klassen er for tæt koblede.

Noter

For bedre at kunne diskutere egenskaberne af gode klasser, vil vi her studere en dårlig klasse. På næste slide vises en forbedret udgave af dette eksempel.

Eksempel på en forbedret klasse.

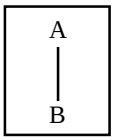
```
Class  regning

feature {NONE}

  Kunde_navn: string;
  Vare_liste: list[varer];

  create(navn: string; varer: LIST[varer]) is
  do ... end;

feature

  udskriv(info_destination: YY) is
  do  ... end;

  -- andre operationer på regninger

end;
```

PS1 -- Opsamling af OOP

Noter

Ideen med denne klasse er at beskrive dataabstraktionen regning. En regning har både tilstand og et antal af operationer, som opererer på tilstanden. Det er disse operationer som udgør klientinterfacet af klassen. Bemærk særligt, at vi nu har en create operation, som opretter en regning.

Interfaces.



En klasse er typisk

- relativt åben i forhold til subklasser, og
- mere lukket i forhold til klienter.

B's klient-interface er typisk en udvidelse af A´s klient-interface.

Eiffel klasser:

Total åbenhed mellem A og B.

Uafhængighed mellem A's og B's klient-interfaces.

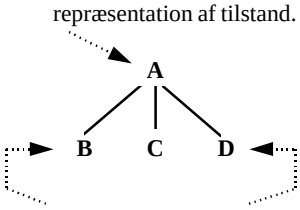
Inducerer typeproblem 2

Noter

Typeproblem 2 vil blive beskrevet senere i dette kapitel.

Interfaceteknikker.

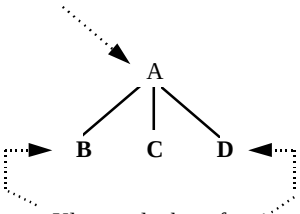
En *Abstrakt klasse* som definerer klient interfacet, men ikke repræsentation af tilstand.



Klasser, der definerer repræsentationen af tilstand samt tilknyttede operationer, men blot afspejler A's klient interface.

- Funktionelt ækvivalente objekter, med forskellig repræsentation kan sam-eksistere.

En konkret klasse, der definerer en dataabstration, men intet klient interface.



Klasser, der hver for sig definerer egne *syn* på dataabstraktionen, som arves fra A.

Syn:
- Udvalget af operationer.
- Teminologi.

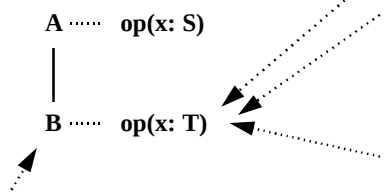
PS1 -- Opsamling af OOP

Noter

Nedarvning.

- Tilføjelse af nye egenskaber til nogle eksisterende egenskaber.
- Redefinition af nogle eksisterende egenskaber.

Kontrolleret redefinition:



Den effektive invariant af klassen B er styrket i forhold til invarianten af A.

- Op har samme antal parametre som A's op.
- Specifikation af op:
  - Prebetingelsen må ikke styrkes.
  - Postbetingelsen må ikke svækkes.
- Signaturen af op:
  - Co-varians: T er en subklasse af S.
  - Contra-varians: T er en superklasse af S.
  - **Eiffel foreskriver co-varians.**

Inducerer typeproblem 1.

Noter

Vi vil også vende tilbage til typeproblem 1 senere i denne forelæsning.

Nedarvningsteknikker (1).

Definition af specialiseringer.

X
|
Y

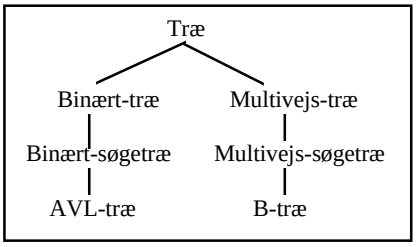
a: X;
b: Y

!!b.create(...);

Objektet refereret af b er et Y-objekt, men det opfattes også som et X-objekt.

Extensionen af Y-begrebet er en delmængde af extensionen af X-begrebet.

Eksempel:



Definition af generaliseringer:

Understøttes ikke af sproglige mekanismer i objekt-orienterede programmeringssprog.

Noter

Nedarvningsteknikker (2).

Formidling af operationer og konstanter som ønskes umiddelbart til rådighed.

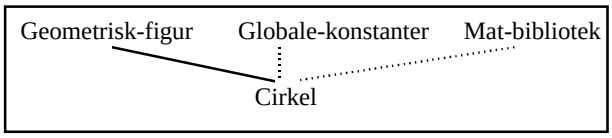
X
⋮
Y

```
a: X; --meningsløst
b: Y

!!b.create(...);
```

I operationerne på klassen Y er egenskaberne i X umiddelbart tilgængelige, som om de var definerede i selve Y.

X er ikke nogen dataabstration. Extensionen af X er meningsløs.

Eksempel:



Muligheden for at have multiple super-klasser er essentiel.

Noter

På denne slide antages, at klassen X indeholde er samling af operationer og/eller konstanter, som ønskes gjort tilgængelig i klassen Y.

Nedarvningsteknikker (3).

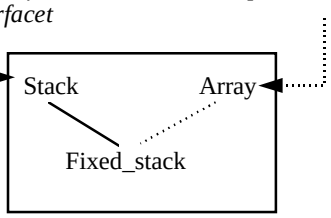
"The mariage of convenience":

Specialisering, som i tilgift bliver
tilført nogle nødvendige mekanismer
til realisering af den ønskede funktionalitet
af den specialiserede klasse.

Eksempel:

Abstrakt klasse som definerer interfacet

Klasse som definerer implementations-mekanismer



Muligheden for at have multiple super-klasser er essentiel.

Noter

"The mariage of convenience" er et specialtilfælde af den anvendelse af nedarvning, som forekommer på den forrige slide.

'Køber' eller 'Arving'?

Problem:
- Klassen B har behov at benytte A-egenskaber.
- Skal B være en klient af A, eller skal B arve fra A?

Arve:

- hvis B *er* en A.
- hvis B ønsker umiddelbar adgang til egenskaberne i A.
  - Ofte relevant hvis A ikke er en dataabstraktion, men en løsere samling procedurer eller konstanter.

- B får adgang til en relativ stor, men potentiel ustabil mængde af operationer, inklusiv repræsentation af data.

Klient:

- hvis B *har* en A-del.

- B får adgang til en relativ mindre, men mere stabil mængde af operationer (repræsentationsuafhængighed).

Noter

Typeproblem 1.
Co-varians.

Typer af operationsparametre varierer på *samme* måde, som de klasser, hvori operationerne befinder sig.

Klasser og nedarvning:

Objekter og operationskald:

```
Aref: A;
Bref: B;
Sref: S;
!!Bref.create(...); !!Sref.create(...);

Aref := Bref;          --Aref betegner et B-objekt

Aref.op(Sref)          --OK idet Aref har statisk type A.
                       --Op i klassen A tager et objekt af typen S som parameter.
                       --Via dynamisk binding anvendes B's op.
                       --En T-operation på et S-objekt er meningsløst.
```

PS1 -- Opsamling af OOP

Noter

Vi antager, at op er redefine'et i forbindelse med, at B arver fra A.

Co-varians problemet's kerne: Vi kalder operationen op på et objekt af typen B. Dette objekt er refereret gennem en variabel, der er erklæret af (har statisk type A). Ud fra et statisk synspunkt er det derfor OK at overføre et S-objekt til operationskaldet. På grund af dynamisk binding bliver det imidlertid operationen op fra klassen B, som kaldes. Dette er en tidsindstillet bombe, idet op fra B kan kalde en T-operation på sin parameter. Kald af en T-operation på et S-objekt er klart meningsløst.

Hvad skal der til for at "demontere" den tidsindstillede bombe? Et run-time check på, at (i vort tilfælde) x faktisk refererer til et T-objekt. For mange run-time checks gør programmer langsomme. Derfor strør compiler-skrivere ikke om sig med sådanne.

I vores Eiffel implementation opstår der en “grim” fejl på udførelsestidspunktet når T-operationen kaldes på S-objektet.

På næste slide viser vi den sammen situation, blot med contra-varians.

Typeproblem 1.
Contra-varians.

Typer af operationsparametre varierer på *modsat* måde, som de klasser, hvori operationerne befinder sig.

Klasser og nedarving:

Objekter og operationskald:

```
Aref: A;
Bref: B;
Sref: S;
!!Bref.create(...); !!Sref.create(...);

Aref := Bref;          --Aref betegner et B-objekt

Aref.op(Sref)          --OK idet Aref har statisk type A.
                       --Op i klassen A tager et objekt af typen S som parameter.
                       --Via dynamisk binding anvendes B's op.
                       --En T-operation på et S-objekt er altid meningsfuld.
```

PS1 -- Opsamling af OOP

Noter

Bemærk at vi stadig benytter den sammen grafiske konvention for nedarvning mellem klasser: Den mest generelle klasse angives øverst på papiret; Den mest specifikke klasse angives nederst.

Problemerne på denne slide er anderledes end på den forrige.

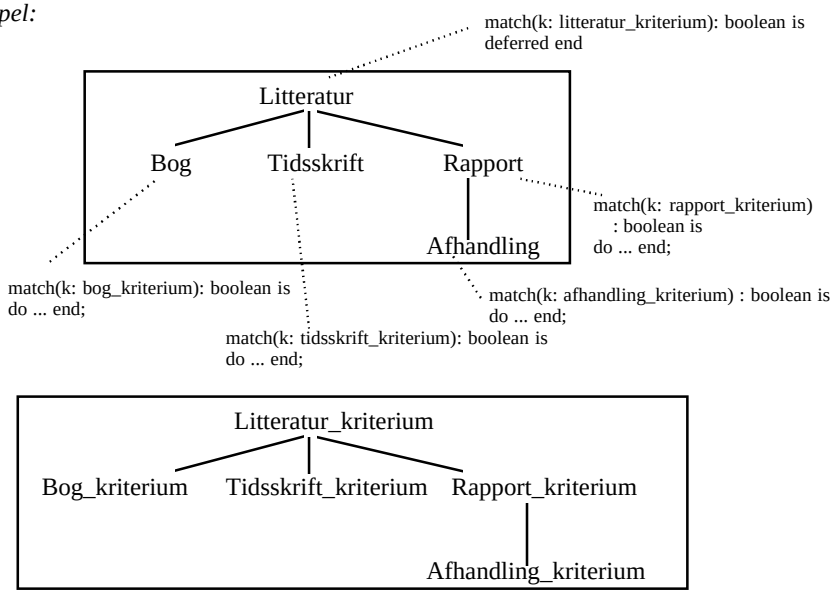
Det er et problem med contra-varians, at Eiffel-reglerne ikke overholdes. Eiffel-compileren checker reglen, og programmet kan ikke oversættes. Men filosofien bag Eiffel kan naturligvis være forkert.

Det er et problem med contra-varians, at det forekommer mere sjældent i praktisk objekt-orienteret programmering end co-varians.

Det er til gengæld *ikke* et problem, at der i operationen op i B kaldes en T_operation på x, som er et S-objekt.

Co-varians i praktisk OO programmering.

Eksempel:



PS1 -- Opsamling af OOP

Noter

Påstanden og dermed pointen på denne slide er følgende: I praktiske sammenhænge har man behov for at berige parametre i takt med at man beriger klasser. I klassehierarkiet i eksemplet specialiseres klassen af litteratur til bøger, tidsskrifter, tekniske rapporter og afhandlinger. I takt hermed har man også behov for at specialisere parameteren af match, som jo er det kriterium, ud fra hvilket match bestemmer, om et givet stykke litteratur matcher de i kriteriet givne egenskaber.

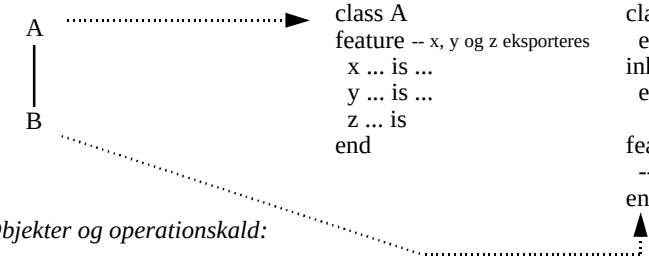
Når der kommer nye egenskaber til i et specialiseret stykke litteratur, opstår der et naturligt behov for nye egenskaber i det *tilhørende* match-kriterium.

Typeproblemerne, som de blev omtalt på sliden om co-varians, optræder (1) hvis en variabel via brug af polymorfi-reglerne refererer et mere specifikt objekt, og (2) hvis der på dette objekt kaldes en operation med et argument, der er et mere generelt objekt. I praksis er det atypisk, at (2) holder.

Vi kan observere at typesammenligneligheden i Eiffel (og de fleste andre OO-sprog) tillader, at en match operation med parameter p kan kalde en mere generel match operation med samme parameter p. (Dette har vi undertiden kaldt imperativ metode-kombination). Ved anvendelse af contra-varians ville dette ikke være muligt.

Typeproblem 2.
Indskrænkninger i klient-interfacet.

Klasser og nedarving:



Objekter og operationskald:

```
Aref: A;
Bref: B;
!!Bref.create(...);

Aref := Bref;

Aref.z          --Statisk set har Aref en z-egenskab.
                --Dynamisk set er denne egenskab ikke tilgængelig på
                --objekter.
```

PS1 -- Opsamling af OOP

Noter

Fra et statisk synspunkt er der ikke noget i vejen med ovenstående program. Eiffel compileren (i version 2.3) opdager ingen fejl. Det ville generelt set kræve omfattende (statisk) analyse af programmet for at erkende, at Aref kan (i dette tilfælde *vil*) referere til et B-objekt. Det er selvfølgelige let at erkende, at B har en mere snæver eksport end klassen A.

I Eiffel 2.3 er der ikke indlagt run-time check, som fanger ovenstående tilfælde. Så på trods af, at z ikke er en del af B's klient-interface, er vi i stand til at referere til z i det B-objekt, som refereres via Aref.