10
Metodekombination og multimetoder i CLOS.

Metodekombination generelt.
Multimetoder generelt.
Kald af en generisk funktion i CLOS.
Specialisering på enkeltobjekter.
Standard metodekombination.
Simpel, indbygget metodekombination.
Brugerdefineret metodekombination.
Eksempler.
Effektiv implementation.

Noter

Metodekombination.

- **Problemet**: Hvorledes kombineres sammenhørende metoder til én overordnet "operation".
  - Løsning 1: ***Imperativ metodekombination***:
    - Programmøren benytter sprogets kontrolmekanismer til at kombinere metoderne.
    - Nogle sprog tilbyder mekanismer der giver tilgang til mere generelle (eller mere specifikke) metoder relativt til en bestemt metode.
  - Løsning 2: ***Deklarativ metodekombination***:
    - Et deklarativ metodekombination afspejler et bestemt samarbejdsmønster mellem metoder.
    - En deklarativ metodekombination er således en *abstraktion* over et samarbejdsmønster mellem metoderne i en generisk funktion.
    - I forbindelse med et operationskald, eller i den overordnede funktion, refereres der til en eksisterende metodekombination.

Noter

Med *sammenhørende metoder* mener vi i CLOS metoderne i en generisk funktion. Dette er de metoder, som har samme navn som den generiske funktion, og hvis parameterlister (i Common Lisp jargon kaldes lambda lister) er kongruente med hinanden.

Den overordnede operation er i CLOS en generisk funktion.

Multimetoder.

I de fleste objekt-orienterede sprog er metoder tilknyttet netop én klasse.

Problem med "single-metoder":

- Hvis en metode opererer symmetrisk på to eller flere objekter (parametre):
  - Klassen af ét af disse objekter skal indeholde metoden.
  - Det udvalgte objekt bliver modtager objekt af "en besked".
  - De øvrige objekter bliver "blot parametre".
  - Asymmetrisk behandling af parametre.

```
(defmethod paint (shape medium)
   (error "Sorry, painting is not possible"))

(defmethod paint ((shape rectangle)
                  (medium screen))
  ...)

(defmethod paint ((shape rectangle)
                  (medium printer))
  ...)

(defmethod paint ((shape color-rectangle)
                  (medium color-printer))
  ...)
```

En løsning: Multi-metoder.

- Man behandler alle parametre ens og symmetrisk.
- Metoder kan ikke være indeholdt i én klasse.
- 'Message passing' metaforen bryder sammen: ikke én modtager af en besked.

PS4 - Metodekombination og multimetoder

Noter

Eksempler på sprog, hvor metoder er tilknyttet netop én klasse er alle de objekt-orienterede programmeringssprog, vi hidtil har været i nærheden af: Smalltalk, C++, Eiffel, Beta, Simula.

Vi vil sige, at en metode er en *single metode*, hvis den netop tilhører én klasse. I de fleste sprog med single metoder vil man opfatte disse som værende indeholdt i en klasse, eller i det mindste fast tilknyttet til netop én klasse. Det er naturligvis et problem for multimetoder at være indholdt i flere klasser. Derfor "svæver" sådanne metoder i CLOS over klasserne. Med andre ord, metoder i CLOS er ikke indeholdt i en bestemt klasse.

Multimetoder er meget velegnede i situationer, hvor nogle veldefinerede kombinationer af argumenttyper skal "håndteres specielt", og øvrige kombinationer skal "håndteres default".

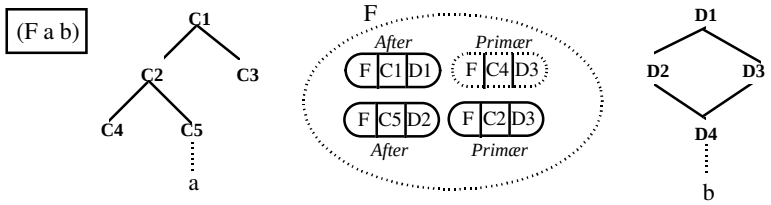
Metodekombination støtter en fragmentering af programmet, hvor én metode håndterer ét tilfælde (som skal kunne indfanges af parameter specialiseringer).

Det er fristende at sammenligne multimetoder med funktionsdefinition på clausal form, som det kendes fra bl.a. ML. I CLOS er det muligt at udfaktorisere en generisk funktion i et antal metoder, hvor parameterlisten definerer under hvilke omstændigheder metoden kan komme i anvendelse.

Oversigt over kald af en generisk funktion i CLOS.

Kald af en generisk funktion på en række objekter:

1. **Udvælg** de *anvendelige metoder*.
   Klassetilhørsforholdet af alle argumenter er af betydning.
2. **Orden** de anvendelige metoder.
   Ordningen sker ud fra *class precedence listerne* af involverede klasser.
3. **Kombiner** de ordnede, anvendelige metoder til en *effektiv metode*.
   Kombinationen sker i forhold til metodernes roller.
4. **Aktiver** den effektive metode.



PS4 - Metodekombination og multimetoder

Noter

De fire punkter ovenfor er de centrale og vigtige skridt, som altid gennemføres når en generisk funktion kaldes i CLOS.

Nedenfor "opskriften" er tegnet et scenario, som er egnet til at diskutere de enkelte skridt i kaldet. Vi har to klassehierarkier (som nok på mere generelt niveau har fælles forfædre, men dem viser vi ikke her). Vi har endvidere fire metoder i en generisk funktion, som hedder F. De fire metoder er multimetoder, idet de alle specialiserer på to klasser: én C-klasse, og én D-klasse. (Bemærk, at dette er en nødvendighed, idet vi jo stiller visse krav til metoderne i en generisk funktion om passende ensartede parameterlister - kongruens). Endelig har vi indtegnet to objekter, kaldet a og b, som hhv. er instanser af C5 og D4. I denne situation kalder vi F på a og b: (F a b).

Alle metoder pånær F på C4 og D3 er anvendelige. Denne metode kan ikke anvendes, idet a ikke er af klassen C4.

De anvendelige metoder ordnes i følgende rækkefølge: (fra mest specifikke til mest generelle):

1. F på C5 og D2
2. F på C2 og D3
3. F på C1 og D1

(Der kunne godt være to metoder på samme kombination af klasser, hvis disse har forskellige roller. Det viser sig unødvendigt at bekymre sig om, hvorledes sådanne to metoder ordnes indbyrdes).

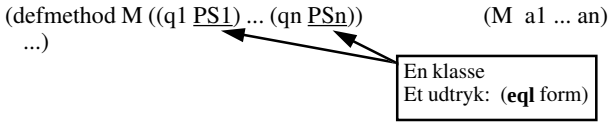
Afhængig af de tre anvendelige metoders rolle, dannes nu en effektiv metode, ved brug af den ønskede metodekombination. Den effektive metode er altså den samlede, resulterende operation som aktiveres, givet typerne af de supplerede argumenter.

Den primære metode er F på C2 og D3, som kaldes og som bestemmer værdien af F. Dernæst kaldes de to after-metoder i følgende rækkefølge (most specific last):

1. F på C1 og D1
2. F på C5 og D5

Hvor angives, hvilken metodekombination, der ønskes? Det er en egenskab af den generiske funktion, som jo defineres af **defgeneric** makroen.

Metode anvendelighed.



Problemet: Er metoden M anvendelig på a1 ... an?

Regel: Metoden M er anvendelig på a1 ... an hvis
For alle i
Hvis PSi er en klasse, og klassen af ai er C:
C <= PS1 (C er klassen PS1, eller en subklasse af PS1).
Hvis PSi er (eql form), værdien af form er *object*.
(**eql** ai *object*).

Noter

Når en parameter specializer, PSi, er på formen (**eql** form) siger vi, at der *specialiseres på et enkeltobjekt*. På denne parameterposition er metoden kun anvendelig, hvis argumentet ai er lig med det objekt, som formen evalueres til.

Bemærk, at form i (**eql** form) evalueres i den leksikalske omgivelse, der er til stede i defmethod sammenhæng, og at formen kun evalueres én gang.

Vi uddyber specialisering på enkelt-objekter på næste slide.

Specialisering på enkelt-objekter.

- Det er muligt at specialisere en parameter i en metode på et enkelt objekt, fremfor en klasse af objekter.
- Metoden er kun anvendelig, hvis det anvendes på et objekt, der er lig med (eql) med parameterspecialiseringen.

Eksempel:

```
(defgeneric coerce (source-object target-type)
  (:documentation "If possible, coerce source-object
                   to target-type."))

(defmethod coerce ((source t) (target-type t))
  ; default metode
  source)

(defmethod coerce ((source vector) (target (eql 'list)))
  ...)
```

```
(defmethod coerce ((source string)
                   (target (eql 'list)))
  ...)

(defmethod coerce ((source string)
                   (target (eql 'char)))
  ...)

(coerce "pip" 'list)  => ("p"  "i"  "p")
(coerce "pip" 'cons)  => "pip"
(coerce #(p i p) 'list)  =>  (p i p)
(coerce "p" 'character)  => #/p
```


Noter

Cons som parameterspecialisering i (eql ‘cons) burde måske dækkes at list tilfældet. Dette er for mig et signal om, at der burde åbnes for kunne anvende parameterspecialiseringer med andre sammenligningsprædikater end eql. Evt. kunne man ønske at anvende prædikater, man selv har defineret.

Man kan opfatte typerne string og vector mv. som klasser, vi enten selv har defineret, eller som predefinerede klasser i CLOS. Faktisk er disse typer Common Lisp typer, som automatisk i CLOS opfattes som klasser. Dette forhold vil vi ikke gå så meget op i i vores sammenhæng.

Oversigt over ordninger.

Klasser	C1 <= C2 *Class precedence order* C1 er mindre generel (mere specifik) end C2
Metoder	M1 <= M2 M1 er mindre generel (mere specifik) end M2. Ordningen er baseret på class precedence lists, og konkret defineret af ordningen af parameter specialiseringslister.
"Hjælpe-ordninger":	
Parameter specialiserings lister	PSL1 <= PSL2 *Leksikografisk ordning* baseret på ordningen af parameter specialiseringer.
Parameter specialisering	PS1 <= PS2 PS1 er mindre generel (mere specifik) end PS2.

Noter

Denne slide giver en oversigt, over de forskellige ordninger som kommer ind i billedet, for at kunne sortere de anvendelige metoder.

Udgangspunktet er ordningen af en klasse, og alle den direkte og indirekte superklasser. Dette er class precedence listen, som blev diskuteret i forrige kapitel.

Ordningen af metoder afgøres ene og alene af en ordning defineret på parameter specialiseringslisterne. Dette er en struktur, der er dannet af alle de påkrævede parameters klassetilhørsforhold (eller specialisering på enkeltobjekter).

Ordningen af parameterspecialiseringslister induceres af en ordning, vi definerer på de enkelte parameterspecialiseringer. Dette er vist på den næste slide. Det viser sig, at den indbyrdes ordning af to parameter specialiseringer i et og alt er defineret af class precedence listen, som vi tog udgangspunkt i i denne diskusion.

Ordningen af parameterspecialiseringslister, og dermed metoder, ud fra den *leksikografiske ordning* baseret på parameterspecialiseringsordninger, er værd at bide sig fast i. Dette betyder nemlig, at den første parameterposition i en metode er mere betydende end de efterfølgende. Dette er helt analogt til, at strengen “abe” er mindre end “zebra”, idet tegnet ‘a’ er mindre en tegnet ‘z’. Hvis to metoders indbyrdes orden ikke kan afgøres via første parameterposition (altså hvis disse parameterspecializers er lig med hinanden) tager vi de efterfølgende parameterpositioner i betragtning (i rekusiv fortolkning, som sædvanlig ved leksikografisk ordning).

Bemærk, at det i CLOS er muligt at påvirke, i hvilken rækkefølge parameterne skal indgå i den leksikografiske orden. Dette gøres i **argument-precedence-order** egenskaben, som kan angives i defgeneric definitioner.

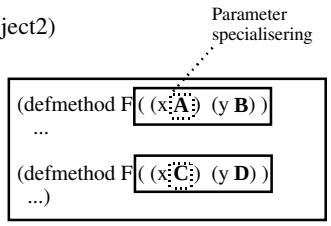
Ordning af parameter specialiseringer.

Defintion.

PS1 <= PS2 hvis og kun hvis en af følgende tre tilfælde er opfyldt:

1. PS1 = C1 og PS2 = C2 og C1 <= C2
2. PS1 = (**eql** object) og PS2 = C2
3. PS1 = (**eql** object1) og PS2 = (**eql** object2)

C1 og C2 betegner klasse navne.



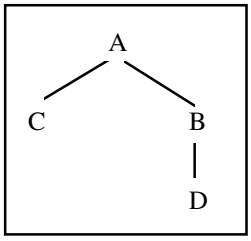
Noter:

- I tilfælde 3 vil object1 og object2 være identiske (idet de to metoder, hvori PS1 og PS2 forekommer, begge er anvendelige).

Noter

Lad klasserne A, B, C og D være defineret som i hosstående klassehierarki. Relativt til kaldet (F c d), hvor d er en D-instans, og hvor c er en C instans, er begge af ovenstående metoder anvendelige.

Hvad er deres indbyrdes ordning? Idet C <= A afgøres rækkefølgen af den første parameterposition. Derved bliver metode 2 mindre generel end metode 1.



Standard metodekombination i CLOS: Roller og aktivering.

Primære metoder

- Vigtigste metode
- Kan bestemme returværdien af en generisk funktion.
- Default metode rolle.

Hjælpemetoder

Before metods

- Udføres før primær metode.
- Kun sideeffekter.

After methods

- Udføres efter primær metode.
- Kun sideeffekter.

Around methods

- Udføres istedet for before+primær+after metoder.
- Afgør værdien af generisk funktion.
- Kan eksplicit kalde before+primær+after metoder.

Kald mest specifikke **around** metode.

Kald alle anvendelig **before** metoder
Mest specifikke først.

Kald mest specifikke **primær** metode

Kald alle anvendelige **after** metoder.
Mest specifikke sidst.

(call-next-method) virker på:

- around metoder
- primær metoder

PS4 - Metodekombination og multimetoder

Noter

Kombinationen af before-metoder, primære metoder og after-metoder i den inderste stiplede ramme ovenfor, kaldes *core frameworket.*

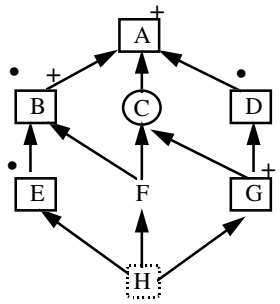
Call-next-method er CLOS primitivet, hvormed man kan lave imperativ (direkte programmørstyret) metodekombination. Primitivet skal benyttes fra around metoder for at aktivere core-frameworket. Med andre ord, det er nødvendigt at aktivere call-next-method fra en eller anden around metode for at aktivere before- primære- og aftermetoder i det mønster, vi diskuterer ovenfor.

Mere præcist beskrevet sker der følgende ved kald af call-next-method:

> Fra en around metode: En around metode aktiverer den around methode, der er den næstmest specifikke around metode, hvis en sådan findes.
>
> Fra en around metode, hvortil der ikke findes en mere generel, anvendelig around metode: Denne around metode aktiverer core frameworket.
>
> Fra en primær metode: Den mest specikke primære metode aktiverer den næstmest specifikke primære metode.

Se endvidere afsnittet “Standard Method Combination” side 1-29 i rapporten, som giver kort og klar besked om ovenstående.

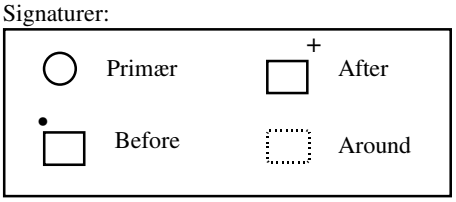
Standard metodekombination: et generelt eksempel.

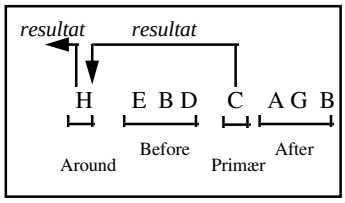


Class precedence list af klassen H:

H < E < F < B < G < C < D < A

Signaturer:



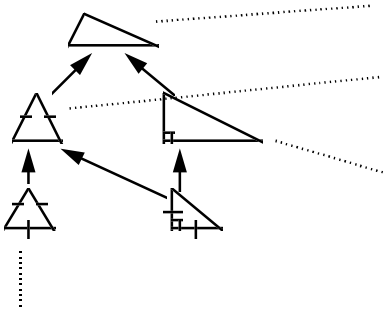


PS4 - Metodekombination og multimetoder

Noter

På denne slide antages, at metoder er knyttet til en enkelt klasse i klassehierarkiet givet ovenfor. Dette kan opnåes ved kun at specialisere ét af de påkrævede parameter i de metoder, som vi har antydet. Vi antager også, at alle de pågældende metoder er anvendelige. Glem alt om multimetoder i forhold til dette eksempel.

Standard metodekombination: trekant eksempel.



(defmethod draw ((tri triangle))
"tegner trekantens tre linier")

(defmethod draw :after ((tri isoceles))
"tegner indikeringer af to lige lange sider")

(defmethod draw :after ((tri right))
"tegner indikeringer af ret vinkel")

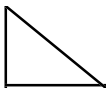
(defmethod draw :after ((tri equilateral))
"tegner indikering på én side")

Noter

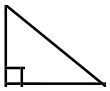
Tegning af ligebenet retvinklet trekant

Class prec list: ligebenet-ret ligebenet retvinklet trekant.

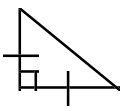
Primær metode tegner trekant



Retvinklets after metode tegner vinkel mærke.



Ligebenets after metode tegner side mærker.



(draw (make-instance 'isocelles-right :side 5))

Standard metodekombination: flere eksempler.

Aktive værdier:

```
(Defclass A ()
  ((value :accessor y)))
```

```
(defmethod y :after ((x A))
   "Denne metode aktiveres efter  'value' er
    aflæst")

(defmethod (setf y) :after (new-value (x A))
   "Denne metode aktiveres efter 'value' er
    overskrevet")
```

"Doven tilstand":

```
(defclass A ()
  ((state :accessor acc
          :documentation
            "state er enten beregnet
             eller pt. uberegnet")))
```

```
(defmethod acc :before ((x A))
   (if (not (calculated (slot-value x 'state)))
       (setf (slot-value x 'state)
             "den rigtige værdi")))
```


Noter

En aktiv værdi er et objekt, hvortil man kan knytte handlinger (procedurer), som bliver aktiveret når værdien manipuleres. Mere konkret, kan man arrangere at der aktiveres en handling når værdien aflæses og/eller når værdien overskrives. Det vises i det øverste eksempel, hvordan dette kan arrangeres i CLOS ved brug af after metoder i standard metodekombination.

I det nederste eksempel illustreres det hvordan en before metode i standard metodekombination kan realisere "doven tilstand". Med doven tilstand mener vi en tilstand, som først bliver beregnet når der er behov for den. I det konkrete eksempel vil before metoden beregne tilstanden på en eller anden måde lige inden den primære metode tilgår den. Jeg har selv brugt netop det ovenstående mønster i en anvendelse, hvor jeg havde en mængde knuder lagret på filer. Nogle knuder er læst op fra filerne, andre er ikke. Hvis jeg tilgår en knude, hvis indhold er på en fil, vil en before metode indlæse filen inden knuden tilgåes via primær metoden.

Simpel metodekombination.

+ and append list max min nconc or progn

Semantikken af simple indbyggede metodekombinationer:

- Lad (F a1 a2 ... an) være et generisk funktionskald.
- Lad *operator* være den benyttede metodekombination i F.
- *operator* er en af **+ and append list max min nconc or progn :**

  1. Around metoder kaldes som ved standard kombination**.**
  2. Flg. kald udføres:
     (*operator* (M1 a1 a2 ... an) ... (Mk a1 a2 ... an))
     hvor M1, M2, ... Mk er alle anvendelige *primære* metoder for F

- Gør kun brug af around og primære metoder
- Before og after metoder kan (og må) ikke bruges.

Noter

Ovenstående viser en metodekombination, som er meget simplere en standard metodekombination. Ideen er at se bort fra before og after metoder, men til gengæld at kombinere alle primære metoder ved brug af en nærmere angivet Common Lisp funktion. Det angives i defgeneric formen, om der skal anvendes en af de simple metodekombinationer, vi diskuterer på denne side, om der skal anvendes standard metodekombination, eller om der skal anvendes en helt tredie metodekombination.

Ordningen af operatorerne i formen

(*operator* (M1 a1 a2 ... an) ... (Mk a1 a2 ... an))

er most-specific-first af de indgående metoder.

Programmør-definerede metodekombinationer.

1. Simpel og begrænset **define-method-combination**
   - Brugeren kan selv definere nye metode-kombinationer, der svarer til simple, indbyggede metodekombinationer.
2. Generel og kompliceret **define-method-combination**
   - definere hvilke metode-roller, som metode-kombinationen understøtter.
   - definerer om der skal være mindst én metode i en rolle.
   - definerer en Lisp-form, som skal udgøre den *effektive metode.*

     Den effektive metode har adgang til:
     - Metoderne i de enkelte roller.
     - Argumenterne, som den generiske funktion er kaldt med.
     - Det generiske funktionsobjekt.

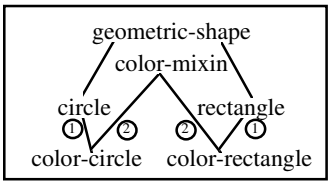
Noter

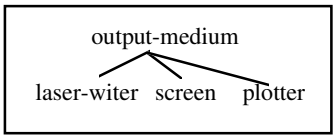
Vi vil ikke gå i detaljer med programmør definerede metodekombinationer på dette kursus.

Der henvises til beskrivelsen af makroen define-method-combination i kapitel 2 af CLOS rapporten, som indeholder en flere-sidet beskrivelse af define-method-combination, inklusive eksempler.

Multimetoder og standard metodekombination: paint eksempel.

```
(defgeneric paint (shape medium)
  :documentation " paint shape on medium" )

(defmethod paint ((shape rectangle) medium)
  ...)

(defmethod paint ((shape circle) medium)
  ...)

(defmethod paint :before ((shape color-mixin) (medium plotter))
  (set-color (color (shape))))

(defmethod paint :before ((shape geometric-shape)
                          (medium output-medium))
  (startup medium))

(defmethod paint :after ((shape geometric-shape)  (medium plotter))
  (advance-paper medium))
```





PS4 - Metodekombination og multimetoder

Noter

Dette eksempel illustrerer både standard metodekombination og multimetoder.

Tallene i klassehierarkierne angiver ordningen af superklasserne i de respektive klassedefinitioner. Vi antager at både color-mixin og geometric-shape (implicit) arver fra standard-object.

Eksemplet viser to primære metoder, som hhv. udskriver et rektangel og en cirkel på et vilkårligt medium. Det ville naturligvis have været muligt at differentiere mellem medier også i primære metoder (men det har vi ikke gjort her).

De to before metoder tager sig af forberedelserne til det egentlige arbejde, som jo altså udføres af en primær metode. Dette forberedende arbejde består hhv. i at starte apparatet på (fra en tænkt dvaletilstand -- ens for alle output medier), og at sætte farvesystemet op på plotteren i forhold til de farver, der måtte være brug for.

Aftermetoden køre papiret frem, så plotteren er klar til næste udskrift.

Eksempel: (paint my-color-rect plotter1). Vi antager, at my-color-rect er en instans af color-rectangle (som navnet også mere end antyder), at at plotter1 er en instans af en plotter. Class precedence list af klassen color-rectangle er

color-rectangle, rectangle, geometric-shape, color-mixin, standard-object, t

I så fald vil følgende metoder blive kaldt i denne rækkefølge:

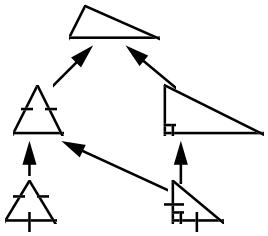
paint :before ((shape geometric-shape) (medium output-medium))

paint :before ((shape color-mixin) (medium plotter))

paint ((shape rectangle) medium)

paint :after ((shape geometric shape) (medium plotter))

Multimetoder: kongruens af trekanter.



```
(defmethod congruent ((tri1 triangle) (tri2 triangle))
   ...)

(defmethod congruent ((tri1equilateral) (tri2 right))
  nil)

(defmethod congruent ((tri equilateral) (tri2 equilateral))
  (eql (side tri1) (side tri2)))

....
```

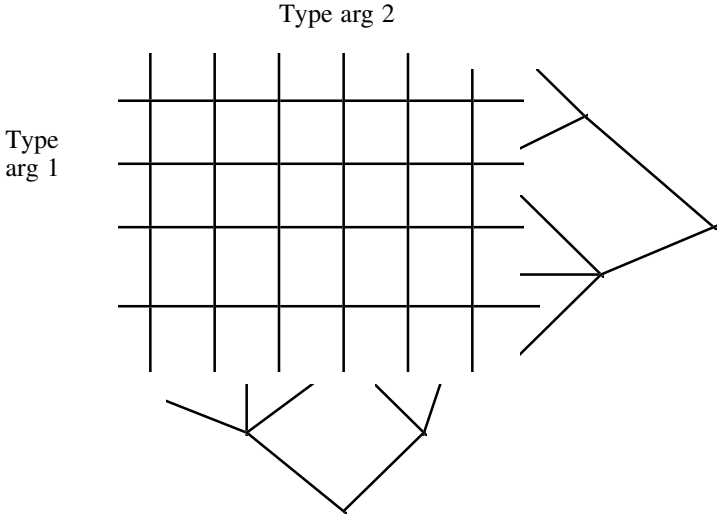
CLOS sorterer mængden af anvendelige *metoder i henhold til parameter listens specificitet. Den mest specifikke (primære) metode udvælges.*

Noter

Bemærk, at 'equilateral' i vores sammenhæng betyder ligesidet.

Ovenstående eksempel viser hvordan en række forskellige tilfælde af kongruens mellem to trekanter kan faktoriseres ud i multimetoder, der tager to trekantsparametre. Det ses, at i nogle tilfælde kan man med det samme melde om "ingen kongruens" (svar: nil).

Man kan sammenligne ovenstående med en tabelstyret tilgangsvinkel. For hvert muligt par af trekantstyper kan vi naturligvis skrive en funktion, der afgør om det pågældende par er kongruente. I nogle tilfælde vil det være hensigtsmæssigt at kunne håndtere funktioner i to eller flere felter sammen. Multimetoder i CLOS tillader sammen med nedarvning i et klassehierarki netop dette. Nedenstående figur forsøger at illustrere denne pointe.



Effektiv implementering af generiske funktioner.

- Ved definition og re-definition af en generisk funktion f og metoder i denne:
  - I en inkrementel og interaktiv omgivelse kan det være for langsomt at foretage systematisk caching af effektive metoder efter hver definition af en funktion.
    - » *Udsæt mest muligt af denne caching til udførelsestidspunktet.*
- Ved kald af generisk funktion f: (f a1 a2 ... an)
  - Lokalisering af den *effektive metode* for f.
    - Det er for langsomt at kombinere metoderne i den generisk funktion ved hvert kald.
      - » *Foretag mest mulig forudberegning og caching på funktionsdefinitionstidspunktet.*
  - Hvis hvert argument a_i kan være af $<=_i$ forskellige typer, kan der findes $<=_1 \cdot <=_2 \cdot \ldots \cdot <=_n$ forskellige effektive metoder.
  - Systematisk caching er derfor udelukket.
- Pragmatisk kompromis (balance mellem tid/plads og effektiv fnk. def/kald):
  - En effektiv metode dannes generelt ved første kald, og *memoiseres*.
  - Håndter ofte forekommende specialtilfælde mere effektivt, om muligt.

Noter

En effektiv implementation af generiske funktioner, i al deres rigdom, er naturligvis alfa og omega for et CLOS system. Vi vil ikke i dette kursus gå i dybden med dette emne, blot kradse lidt i overfladen af problemet.

Bemærk den indbyrdes modstrid mellem hvilke bidrag der udføres på funktionsdefinitionstidspunkt, og hvilke bidrag der udføres på funktionskaldstidspunktet (i de to ønsker mærket med >>). Dette leder naturligt til en eller anden "trade off" mellem effektiv, incrementel programmeringsstil, og effektiv udførelse af et program.

En mulig og typisk løsning er at anvende en *hashtabel*, som mapper klasser (eller tupler af klasser i tilfælde af multimetoder) til en effektiv metode. Problemet er at holde en sådan hashtabel opdateret i forhold til inkrementelle programændringer. Et andet problem (som omtalt ovenfor) er, at tabellen bliver så stor, at det næppe er realistisk at beregne den fuldt ud, fordi en sådan beregning tager tid, og resultatet fylder meget. Ved *memoisering* husker vi en allerede beregnet effektiv metode, således at den ikke skal genberegnes (kombineres mv.) ved kald på argumenter af samme typer (hvilket nok er mere typisk end man først skulle tro).

Det forholder sig således at metode re-definition er forholdsvi let at håndtere effektivt på funktionsdefinitionstidspunkt. Dermed mener vi, at det er overkommeligt for de fleste funktioner (uden alt for mange parametre, og uden alt for mange hjælpemetder) at chace de forskellige effektive metoder. Men det kræver naturligvis lager.

Redefinition af en klasse er langt mere problematisk. Hvis man ændrer på superklasserne af en klasse, ændres med ét slag anvendelighedsforholdet af alle metoder på alle subklasser af den berørte klasse. For "højtplacerede klasser" vil dette tage lang tid at håndtere. Lige så alvorlige problemer opstår hvis man ændrer (sletter, tilføjer, modificerer) slots i en klasse, idet CLOS jo forsøger at opdatere alle eksisterende objekter af den berørte klasse. (Dette så vi på i forrige kapitel).

Litteratur: "Efficient Method Dispatch in PCL" af Gregor Kiczales og Luis Rodriguez, 1990 ACM Conference on Lisp and Functional Programming.