9
Introduktion til CLOS.

Klasser
Metoder
Generiske funktioner
Slots
Nedarvning
Class precedence lists
Klasse-redefinition og objektopdatering
Ændring af objekters klassetilhørsforhold
Eksempler

PS4 - CLOS

Noter

I denne forelæsning vil vi introducere programmeringssproget CLOS (the Common Lisp Object System). Dermed lader vi op til næste forelæsning, og en senere forelæsning, om hhv.

Metodekombination og multimetoder, og

Metaobjekt protokoller i CLOS.

Når vi taler om CLOS rapporten mener vi "Common Lisp Object System Specification X3J13", document 88-002R, af Bobrow et al. Rapporten er bl.a. publiceret i Sigplan Notices vol 23, special issue, September 1988. Rapporten indeholder to kapitler. Det første inholder en emnemæssig opdelt beskrivelse af CLOS. Det andet indeholder en bekrivelse af de makroer og (generiske) funktioner, som udgør CLOS. Hvis man skal programmere i CLOS, er det andet afsnit meget nyttig som reference manual.

Klassebegrebet i CLOS.

- Klasser defineres ved brug af makroen **defclass**.
- En klasse er karakteriseret af
  - klassens navn.
  - en liste af superklasser.
  - en mængde af *slot egenskaber:*
    - initialværdi.
    - argumentnavn til initialisering.
    - navn på accessor funktion.
    - allokering, type, dokumentation, ...
  - en mængde af *klasse egenskaber*:
    - dokumentation.
    - metaklasse.
  - klassens operationer (metoder) er *ikke* indeholdt i selve klassen.
- En klasse instantieres af **make-instance**.

```
(defclass rectangle (geometric-shape)
 ((height :initarg :start-height
          :initform 5
          :accessor rectangle-height)
  (width :initarg :start-width
         :initform 8
         :accessor rectangle-width)))
```

```
(defclass circle (geometric-shape)
 ((radius :initarg :radius
          :initform 6
          :reader radius)))
```

```
(setq r1 (make-instance 'rectangle
 :start-height 10 :start-width 15))
```

PS4 - CLOS

Noter

En klasses tilstand (attributter) er beskrevet af en række såkaldte slots i CLOS. En slot har et navn og en række egenskaber. En vigtig egenskab er

:initform *expr*

hvor *expr* er et Lisp udtryk, hvis værdi bliver slottens startværdi. Det er også muligt at angive et argumentnavn, som kan refereres under instantieringen, således at man på instantieringstidspunktet kan give slotten et navn:

:initarg *:navn*

Man kan angive en type af hver slot, men det er ikke intensionen at dette skal give sig udslag i statisk typing af et CLOS program. Der er snarere tale om et hint til Lisp systemet om effektiv lagring af slotværdier.

Ud over de ovenfor nævnte klasse-egenskaber, findes også en klasseegenskab, som kaldes *default initialiserings argumenter*. Hvis der er givet et initialiseringsargument ifm. instantieringen, er det denne værdi der initialiserer et slot; hvis ikke, tager man hensyn til default-initialiserings argumenterne i klassen; hvis vi endnu ikke har fundet noget, tages der hensyn til initial-værdien, som givet ved slotegenskaben initform.

Alt i alt er initialiseringprocessen af slots ganske kompliceret. Se CLOS rapportens afsnit “Object creation and initialization” og bliv overbevist! At initialisering af objektets tilstand er et vigtigt emne er kendt fra mange andre sprog (også C++ og Eiffel). Emnet er så vigtigt at det retfærdiggør en betydelig kompleksitet, for at befordre en tilpas fleksibilitet og rigdom for programmører i sproget.

Selve CLOS apparatet har ikke noget at tilbyde omkring *information hiding*, altså om det at skjule slots for klienter og subklasser. Synlighedsaspektet kan derimod håndteres gennem package begrebet i selve Common Lisp. Kort fortalt opdeles navnerummet i Common Lisp i “packages”. Nogle navne er interne i en pakke; andre eksporteres til andre pakker. Common Lisp bogen indholder et kapitel om dette, hvis man er interesseret.

Metodebegrebet i CLOS.

- Metoder defineres ved brug af **defmethod** makroen.
- En metode karakteriseres af:
  - en generisk funktion, hvortil metoden er knyttet.
  - en *rolle*, der udtrykker hvordan metoden spiller sammen med andre metoder i sin generiske funktion.
  - en parameterliste.
  - en *anvendelighedsbetingelse*, som udtrykker hvornår metoden er anvendelig ift. aktuelle parametre.
  - en krop.

```
(defmethod paint ((shape rectangle)
                  medium)
  (draw-box (rectangle-height shape)
            (rectangle-width shape)
            medium))
```

```
(defmethod paint ((shape circle)
                  medium)
  (draw-circle (radius shape)
               medium))
```


Noter

Anvendelighedsbetingelsen, som omtalt ovenfor, er integreret i parameterlisten. I Lisp jargon kaldes en parameterliste for en *lambda liste*. Parameterlister i Common Lisp indeholder mange muligheder, såsom optionelle parametre, rest parametre (ala Scheme's dot-notation i parameterlisten) og nøgleordsparametre (som ikke overføres pr. positionel korrespondance). Vigtigst af alle Common Lisp parametre er dog de såkaldt *påkrævede parametre*, som syntaktisk set svarer til "almindelige parametre", som vi kender dem fra Scheme funktioner. Ovenfor er både shape og medium påkrævede parametre.

Parameterlisten i CLOS kaldes en *specialiseret lambda liste*. Den opstår ved at de påkrævede parametre kan associeres med et klassenavn. Vi ser ovenfor at shape parameteren i den øverste metode er associeret med 'rectangel' og at shape i den nederste metode er associeret med 'circle'. Medium parameteren er ikke eksplicit associeret med nogen klasse; hermed har vi implicit associeret denne parameter med den mest generelle klasse i CLOS systemet (som kaldes t). Alt dette betyder, at de pågældende metoder kun er anvendelige, hvis deres tilhørende generiske funktioner anvendes på hhv. et rektangel objekt og et cirkel objekt som 1. parameter. Vi vender tilbage til anvendelighedsbegrebet senere.

Hvis vi kun kunne associere en metode med én klasse, ville vi være på kendt grund. Men i CLOS kan metoder associeres med to eller flere klasser. Dette giver en pæn symmetri i de påkrævede parametre, i modsætning til de fleste andre objekt-orienterede sprog, hvor ét objekt (én parameterposition) er dominerende. Vi taler i CLOS om *multimetoder*, som vi vender tilbage til i næste forelæsning.

Der er ikke eksplicit defineret nogen *rolle* af ovenstående metoder. Dette betyder, at de får tilskrevet rollen *primære metoder*. Vi kunne definere en metode i den generiske funktion paint med rollen :after på følgende måde:

```
(defmethod paint :after (shape medium)
  ...)
```

Rollen har betydning for *metodekombination*, som vi vender tilbage til i næste forelæsning.

Bemærk hvordan vi i kroppen af de to metoder kalder de automatisk genererede generiske funktioner i klasserne circle og rectangle.

Generiske funktioner i CLOS.

- En generisk funktion defineres ved brug af makroen **defgeneric.**
- En "generisk funktion" er et modstykke til "polymorfe funktioner" fra andre sprog.
- En generisk funktion "indeholder" en række metoder (metoder af samme navn som den generiske funktion).
- Alle metoder i en generisk funktion skal være *kongruente* (have passende ens parameterlister).
- Metoder aktiveres gennem kald af en generisk funktion.
- En generisk funktion kan erklæres *eksplicit*, eller *implict* ved metodedefinition.
- Et kald af en generisk funktion kan ikke skelnes fra et kald af en konventionel Common Lisp funktion.

```
(defgeneric paint (shape medium))
```

```
(let ((r (make-instance 'rectangle
                        :start-height 15 'green
                        :start-width 20))
      (screen  some-screen ))
  (paint r screen))
```

PS4 - CLOS

Noter

Man kan syntaktisk definere metoderne i en generisk funktion som en del af defgeneric formen. Men det er ikke særlig almindeligt at gøre det.

En generisk funktion kan tilskrives en række egenskaber. Disse er:

> Dokumentation.
>
> Klassen af det generiske funktionsobjekt. Dette hænger sammen med, at en generisk funktion ligesom en klasse og en metode faktisk er objekter i et kørende CLOS system. Klassen, som instantieres for generiske funktioner kan styres via denne egenskab.
>
> Klassen af alle metoder i den generiske funktion.
>
> Den anvendte metodekombination.

Kongruente metoder: For at kunne "spille sammen" skal metoderne og selve den generiske funktion være passende overensstemmende med hensyn til parameterlisterne. I CLOS rapporten defineres, hvad det vil sige at lambda-listerne af metoder og genriske funktioner er kongruente. Det væsentlige er, at der skal være det samme antal påkrævede parametre, men desforuden er der en række regler for rest, key og optionelle parametre. Der henvises til til CLOS rapporten for detaljer.

“Message passing” kontra funktionskald.

Konventionel funktionskald:

(f obj1 obj2 ... objn) ~
(apply (symbol-function 'f)
(list obj1 obj2 ... obj*n*))

Message passing:

(send mes obj1 obj2 ... objn) ~
(apply (method-lookup (class-of obj1) mes)
(list obj2 ... obj*n*))

Generisk funktionskald:

(f obj1 obj2 ... objn) ~
(apply (method-specified-by
f (class-of obj1) (class-of obj2) ... (class-of obj*n*))
(list obj1 obj2 ... obj*n*))

PS4 - CLOS

Noter

Denne slide illustrerer underliggende forskelle i et objekt-orienteret system der anvender message passing og generiske funktioner. Endvidere sammenlignes med kald af en konventionel Lisp funktion.

Observationen fra eksisterende Lisp systemer med objekt-orienterede overbygninger er, at det er uhensigtsmæssigt at anvende både send-metaforen og konventionelle funktionskald i samme program.

Bemærk, at i det konventionelle funktionskald hentes lambda udtrykket med symbol-function, som returnerner indholdet af en speciel egenskab, forskellig fra værdien, af et symbol. Dette er en afgørende forskel mellem Common Lisp og Scheme. I Scheme kan en værdi jo helt fint være en funktion.

Mere om slots.

- Slots er ikke leksikalsk tilgængelige fra metoder.
- Slots kan kun tilgåes på to måder:
  - via automatisk definerede accessor generiske funktioner.
  - via funktionen **slot-value**.
- Accessor generiske funktioner:
  - reader
  - writer
  - accessor
- Tilgang til slots via generiske funktioner sikrer at klienter bliver mindre afhængige af objektets datarepræsentation.
- Slot-value bør kun bruges i normal programmering, såfremt man eksplicit ønsker at forhindre metodekombination i at finde sted.

```
(defclass  c ()
  ((s :accessor s-acc :type integer )))

(defvar s 7)

(defmethod m ((x c))
    (list s (slot-value x 's)  (s-acc x)))


 (let ((ac (make-instance 'c)))
   (setf (s-acc ac) 5)
   (m ac)))    =>   (7 5 5)
```

```
(defclass c ()
  ((s :reader r)
   (q :writer w)
   (u :accessor y)))

(let ((ac (make-instance 'c)))
  (setf (y ac) 5)
  (w 6 ac)
  (list  (r ac) (y ac)))
```

PS4 - CLOS

Noter

I det øverste eksempel har vi en slot i klassen c, og en global variabel (defineret ved brug af defvar i Common Lisp). Det illustreres, at selv om s bruges leksikalsk i en metode på klassen c, får vi ikke af denne grund hånd på slot s i klassen c; Vi får derimod fat i den globale variabel, som vi har defineret med defvar.

I det nederste eksempel illustreres readers, writers og accessors i en klasse, som vi igen blot kalder c. Bemærk formen

(setf (y ac) 5)

som er et assignment-lignende procedurekald på slottet, som har accessor y i objektet ac. Navnet på den generiske funktion, som skriver 5 i slottet med accessor y er pr. konvention

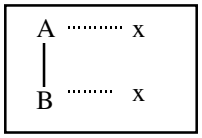
(setf y)

Setf er Common Lisps pendent til Scheme's set!

Common Lisp udnytter dualiteten mellem aflæsning og skrivning på følgende måde: Hvis (acc y) er funktion, som aflæser en bestemt bestanddel af y (eksempelvis (car y)), så er (setf (acc y) x) den tilsvarende *mutator*, som ændrer den samme bestanddel til at have værdien x. Det ganske tilsvarende har man altså overført til CLOS.

Nedarvning.

- CLOS understøtter multipel nedarvning.
- Konflikter og tvetydigheder, som opstår ifm. nedarvning, løses ved brug af *class precedence listen.*
- Class precedence listen af en klasse C er
  - en bestemt, total ordning af C og alle dens superklasser, hvori
  - C og alle dens superklasser optræder netop én gang.
- Nedarvning af metoder: Det er mere naturligt at tale om *metode anvendeligehed* end om nedarvning.
- Nedarvning af slots:
  - Mængden af slotnavne tilgængelige i en klasse C er foreningsmængden af slotnavne i C og alle dets superklasser.
  - Antag at x er en slot i både klassen A og B og at B arver fra A.
    - slot-x's egenskaber i en instans af B er en *kombination* af egenskaberne for slottene x i A og B



I en instans af B er der kun én slot med navn x

PS4 - CLOS

Noter

Håndteringen af to slots med samme navn, som illustreret ovenfor, er interessant i forhold til andre sprog vi kender. I Eiffel gør man et stort nummer ud af renamings for at takle problemet. I C++ skal man fra klienter eksplicit angive, hvilken af de to "slots" man mener. I CLOS, derimod, smelter de to x-er sammen til én slot.

De forskellige slot egenskaber kombineres forskelligt for slots af samme navn:

Allokering: Den meste specifikke allokering bliver gældende.

Initialform: Den mest specifikke initform bliver gældende.

Type: Konjunktionen af alle typer gælder.

Initialargumenter: Foreningsmængden af alle initialargumenter kan benyttes for denne slot.

Dokumentation: Den mest specifikke dokumentationsstreng af slots med samme navn bliver gældende.

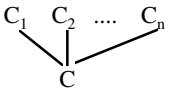
Iøvrigt bliver alle slotegenskaber ordnet efter class precedence listen, hvori de ens navngivne slots befinder sig. Når vi ovenfor siger "Den mest specifikke..." er det relativt til denne ordning af slots.

Class precedence listen.

(defclass C (C_1 C_2 ... C_n)
...)

1. En klasse er mindre generel end dens superklasser
2. En klasse definition bestemmer precedence ordenen af dens superklasser

Notation:
X < Y
"klassen X er mindre generel end klassen Y".



$C < C_1$
$C_1 < C_2$, ... $C_{n-1} < C_n$

- Class precedence listen af en klasse C er en bestemt total ordning (om nogen), som er konsistent med de partielle ordninger, bestemt af ovenstående to regler.
  - Ingen konsistent ordning: der er en fejl i klassehierarkiet.
  - To eller flere konsistente ordninger: Der foretages et deterministisk valg.

PS4 - CLOS

Noter

Givet en mængde af ordninger, hver på formen X < Y (klassen X er mindre generel end klassen Y). Vi antager, at der ikke er selvmodsigelser i mængden (mængden er konsistent). Denne mængde af ordninger kan gøres til genstand for *topologisk sortering*. I denne form for sortering udvælges iterativt den klasse, som ikke har nogen mindre generel klasse i forhold til mængden af bidrag. Hvis dette er klassen C fjernes alle bidrag på formen C < X fra mængden, og iterationen fortsættes. Dette illustreres gennem et eksempel i noterne til næste slide.

Hvis der på et tidspunkt i sorteringsprocessen er to eller flere muligheder for valg af klasse, foretages et deterministisk valg. Reglen er:

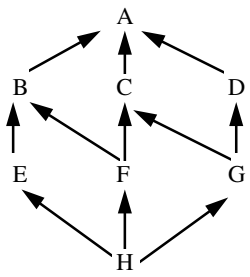
> Hvis der er mere end én klasse, som ikke har mindre generelle klasser i mængden, vælges den af klasserne, som har en direkte *subklasse* mest til højre i den del af klasse precedence listen, som hidtil er dannet.

Hvis der på et tidspunkt ikke kan findes nogen klasse, der kan udvælges, men mængden af bidrag er ikke-tom, er der en fejl i klassehierarkiet. Vi siger, at der en *inkonsistens* i bidragene til ordningen.

Hvis vi udelukkende bruger *enkeltnedarvning*, er ovenstående helt trivielt, idet class precedence listen af C bliver den lineære liste af klasser, startende med C, og sluttende med den mest generelle klasse.

Der må naturligvis ikke være løkker i grafen, som dannes af klassehierarkiet. Hvis der skulle være det, kan vi ikke konstruere class precedence listen.

Class precedence lists: Et eksempel.



B < A C < A D < A

E < B F < B G < C
B < C C < D

H < E
E < F
F < G

Class precedence list af klassen H:

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

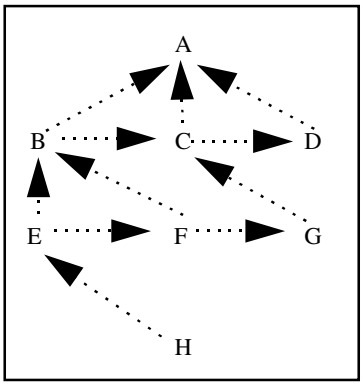
H < E < F < G < B < C < D < A

Problem:
Hvilken af klasserne G og B skal regnes for den mindst generelle i den totale ordning?

PS4 - CLOS

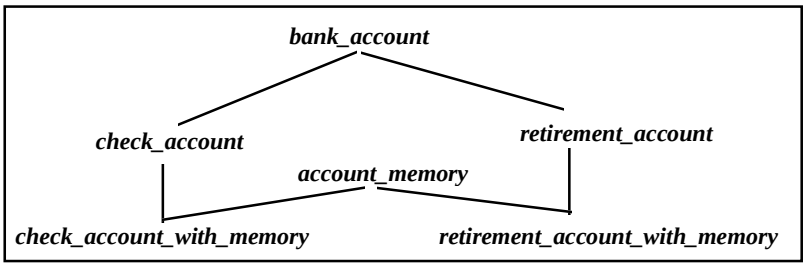
Noter

På denne slide vises et stilistisk klassehierarki, og alle de bidrag til ordningen, som forekommer ud fra reglerne. Disse bidrag danner en partiel ordning. Da den er konsistent kan vi via topologisk sortering, og via anvendelse af det deterministiske valg (én gang) danne en total ordning, vist forneden.



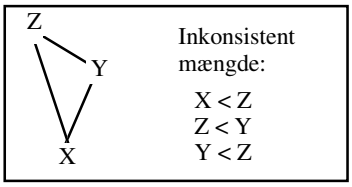
Med det formål at illustrere topologisk sortering laver vi en ny graf, med de samme knuder som i klasshierarkiet. Hvert af primitive bidrag til ordningen giver en kant i den nye graf. Hosstående graf er resultatet, hvis vi anvender denne "transformation" på grafen ovenfor. Topologisk sortering forløber nu ved at vælge en klasse, som ikke (i den nye graf) har en indgående kant. Det er klart H. Vi fjerne nu H og alle kanter fra H til andre knuder. Dernæst gentager vi udvælgelsen efter samme princip. Derved får vi klassen E. På samme måde vælges dernæst F. Således fremskredet er vi nu i en situation hvor vi både kunne vælge B og G, idet hverken B eller G har indgående kanter. Vi vælger B ifølge reglen omtalt på forrige side. Gennemfør selv resten af argumentationen for ovenstående class precedence list.

Flere eksempler på class precedence lists.



Class precedence list for check_account_with_memory:

check_account_with_memory <
check_account <
bank_account <
account_memory



Inkonsistent mængde:

X < Z
Z < Y
Y < Z

PS4 - CLOS

Noter

Class precedence list i bankkonto klassehierarkiet er ikke entydigt bestemt. Den anden mulighed er

check_account_with_memory <

check_account <

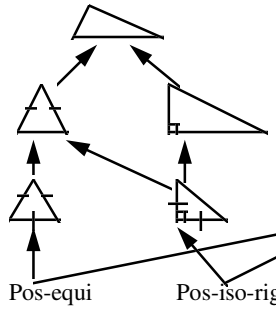
account_memory <

bank_account

Det er dog ikke denne, som vælges ifølge reglen. Iøvrigt er dette eksempel en smule vildledende, idet der altid er en fælles top i et klassehierarki i CLOS, nemlig klasserne standard-object, og en klasse der (tro det eller la' vær') hedder t.

Er det vanskeligt at konstruere et klassehierarki, hvori der optræder en inkonsistens? Nej, det er det ikke. Vi viser et eksempel nederst til venstre. Bemærk, at et sådant klassehierarki både er formelt forkert, men også i praksis er det sygt. Hvorfor lade X arve direkte fra Z når X arver fra Z gennem Y?

Eksempel på klasser: mixin og aggregate klasser.



(**defclass** positioned-object ()
((x :accessor x :initarg :x)
(y :accessor y :initarg :y)
(rot :accessor rotation :initarg :rotation))
(documentation "..."))

Positioned object

Pos-equi

Pos-iso-right

(defmethod displace ((obj positioned-object) x y)
...)

(defmethod rotate ((obj positioned-object) angle
(p point))
...)

(defmetod corner-list ((tri pos-iso-right))
"Returnerer koordinater for trekantens tre hjørner")

Corner-list burder være mere generelt defineret end på klasen pos-iso-right, men dette volder problemer i det viste hierarki!

PS4 - CLOS

Noter

Vi viser her et eksempel, hvor et hierarki af trekanter (vist i billeder ovenfor) suppleres med en klasse, som gør det muligt at positionere en trekant i et koordinatsystem.

Klassen positioned-object er en mixin klasse, idet den kun kan instantieres når den er blandet med en mere substantiel klasse. Klassen indeholder tilstrækkelig information til, at en trekant er placeret i et koordinatsystem. X,y koordinaten af alle vinkelspidser er en mulighed, men mindre information (færre punkter) er nok at foretrække.

I det ovenstående karakteriseres trekanter i det egentlige trekanthierarki udelukkende af side- og vinkelstørrelser.

Klasserne Pos-equi og Pos-iso-right er eksempler på 'aggregate classes', hvilket pr. definition er klasser i CLOS, som blot "aggregerer" to superklasser, men ikke indeholder slots mv.

Den 3. parameter af rotate er omdrejningspunktet.

Opdatering af objekter efter klasse-redefinition.

- **Situation**: I et Lisp system er det muligt at opdatere programbeskrivelsen uden at nedlægge eller nulstille tilstanden af en programudførelse.
- **Problem**: Hvad sker der med objekter, hvis klasse-definitioner redefineres?
- **Svaret** i CLOS:
  - Objekterne opdateres af CLOS uden at skifte identitet.
  - Opdateringen sker senest på det tidspunkt, hvor et slot i et påvirket objekt læses eller skrives.
  - Nogle slots tilføjes, nogle modificeres, og andre slettes.
  - To trin i opdateringsprocessen:
    - Strukturen af objektet tilpasses den redefinerede klasse.
    - Nye slots opdateres ved kald af den generiske funktion ***update-instance-for-redefined-class***.
  - Det er muligt for programmøren at påvirke opdateringsprocessen ved at definere metoder i ***update-instance-for-redefined-class***, typisk :before eller :after metoder.

Noter

Ligesom i Scheme kan vi i Common Lisp og CLOS opdatere programmet (klasser og generiske funktioner, herunder metoder) uden at miste objekter eller dynamisk tilstand. Den mest dramatiske ændring sker ved modifikation af eksisterende klasser, hvoraf der eksterer en stor mængde instanser.

Bemærk at i mere konventionelle, objekt-orienterede sprog er dette ikke et emne, som vi bekymrer os ret meget om. Når vi opdaterer programmet er programudførelsen for længst afbrudt, og der er ingen mulighed for at lade (evt modificerede) objekter overleve relativt til den nye programbeskrivelse. Hvis objekterne er lagret på et persistent medium (på filer, i en objekt-orienteret database, eller lignende) kan situationen dog være anderledes. I denne situation er man nødt til at bekymre sig om den problematik, vi diskuterer på denne slide.

Metoder i den generiske funktion update-instance-for-redefined-class tager fire parametre:

> Instansen som opdateres.
>
> En liste af slotnavne, som findes i den opdaterede klasse men ikke i den oprindelige klasse.
>
> En liste af slotnavne, som findes i den oprindelige klasse men ikke i den opdaterede klasse.
>
> En property liste (n1 v1 n2 v2 ...) over slotnavne og værdier af slots som "smides bort".

Rustet med disse information kan man enten vælge at skrive en ny primær metode i update-instance-for-redefined-class, eller mere typisk, at lave en after metode, hvor man tilføjer passende handlinger, som påvirker den opdaterede instans. Sidstnævnte kan være nødvendigt, idet man ikke kan forvente, at en standard opdateringsstrategi vil virke i alle situationer, hvor man redefinerer en klasse.

Se endvidere afsnittet *Redefining Classes* side 1-43 i CLOS rapporten, som på udmærket vis forklarer klasse-redefinition problematikken, og CLOS' løsninger derpå.

Ændring af klassen af eksisterende objekter.

- **Situation**: Der eksisterer en mængde M af objekter af klassen C.
- **Problem**: Vi ønsker at ændre klassen af objekterne i M til at være C', uden at objekterne skifter identitet.
- **Svaret** i CLOS:
  - Der findes en funktion (***change-class*** instance new-class), der kan aktiveres af programmøren.
  - Klasseændringen sker i to trin. I sidste trin kaldes ***update-instance-for-different-class*** , som initialiserer nye slots relativ til C'.

Eksempel:

```
(defclass position () ())

(defclass x-y-position (position)
  ((x :initform 0 :initarg :x)
   (y :initform 0 :initarg :y)))

(defclass rho-theta-position (position)
  ((rho :initform 0)
   (theta: initform 0)))
```

```
(defmethod update-instance-for-different-class
          :before ((old x-y-position)
                   (new rho-theta-position) &key)
  (let ((x (slot-value old 'x))
        (y (slot-value old 'y)))
    (setf (slot-value new 'rho) (sqrt (+ (* x x) (* y y))))
    (setf (slot-value new 'theta) (atan y x)) ))

(setq p1 (make-instance 'x-y-position :x 2 :y 5))

(change-class p1 'rho-theta-position)
```

PS4 - CLOS

Noter

Problemet, som vi diskuterer på denne side opstår hyppigt i "rigtige" applikationer. Lad os, som et eksempel, kort studere en en klasse, som repræsenterer 'studerende på basisuddannelsen på AUC' i et studenterregistreringssystem. På et tidspunkt skal de fleste af disse objekter skifte klasse til 'studerende på en overbygningsuddannelse'. (Objekter, som ikke tager denne transistion, skal sikkert skifte til 'arbejdsløs'). Hvis ikke man har muligheden for at lade eksisterende objekter skifte klasse, kan det være vanskeligt at håndtere denne tidsmæssige udvikling. Det vil være meget unaturligt og problematisk at nedlægge objekterne af typen 'studerende på basisuddannelsen på AUC' og oprette tilsvarende instanser af 'studerende på en overbygningsuddannelse'. Årsagen er, at objekterne i så fald vil skifte identitet, hvilket er i klar modstrid med forholdene i referentsystemet (virkeligheden).

Eksemplet ovenfor er taget fra CLOS rapporten, kapitel 2.

Se endvidere afsnittet *Changing the Class of an Instance,* side 1-46 i CLOS rapporten, som giver kort og klar besked om denne facilitet.