12
Metaobjekt protokoller i CLOS.

Begreber og problemer.
Sprog designrum.
Niveauer i CLOS.
Programobserverende protokoller.
Programskabende protokoller.
Sprogudvidende protokoller.
Eksempler.

Noter

Begreber og problemer.

- Et *reflektivt system* er et system som indeholder strukturer, der repræsenterer sig selv.
- Et CLOS program er et refleksitvt system, idet der findes objekter (metaobjekter) der repræsenterer selve CLOS programmet.
- Et programmeringssystem baseret på metaklasser (klasser som beskriver program-elementer) er én mulig *refleksiv arkitektur*.
  - I en arkitektur baseret på metaklasser er der en relativ klar adskillelse mellem beregninger på applikations-objekter (hørende til det eksterne problemområde) og refleksive beregninger (hørende til det det selvcentrerede problemområde).
- En *metaobjekt-protokol* er et interface af generiske funktioner, som gør det muligt for en programmør at tilgå og modificere programmers egenskaber samt at ændre sprogets implementation.
  - Metaobjekt-protokollen afspejler omhyggeligt udvalgte *åbninger* i CLOS's implementation. Åbningerne har defaults (metoder og generiske funktioner på standard metaobjekterne). Åbningerne kan "genudfyldes" med primære metoder ,eller "suppleres" med before/after metoder.
  - De generiske funktioner arrangeres i *sub-protokoller*, som hver kontrollerer et bestemt aspekt af sproget.
- Et specielt CLOS problem: Bagud sammenlignelighed.
  - Hvordan forener man en mængde af sprog i nær familie, som dog indbyrdes er forskellige på det syntaktiske og det semantiske plan?

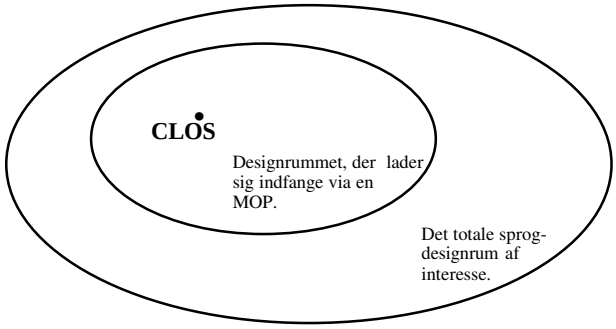
PS4 - Metaobjekt Protokoller

Noter

Artiklen "Concepts and Experiments in Computational Reflection" (OOPSLA 87, Sigplan Notices 22, 12) diskuterer bredt hvad refleksion er, samt de basale begreber (mv.) der kort er nævnt på denne slide.

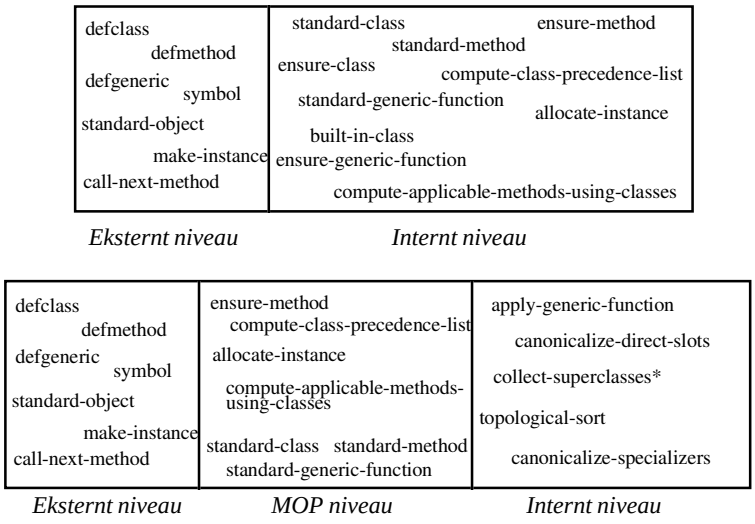
Sprog designrum.

- Et *sprog designrum* er en mængde af programmeringssprog, der har en eller anden mængde af fælles egenskaber.
- Et sprog designrum kan defineres af *et enkelt sprog* (et punkt), som er forsynet med en *metaobjekt-protokol*.



Noter

Niveauer i CLOS.



PS4 - Metaobjekt Protokoller

Noter

Det *eksterne niveau* svarer til den del af sproget, som er synlig og brug for for programmøren. Det er med andre ord sprogets ansigt udaftil, eller den del af sproget, som befinder sig foran scenen.

Det *interne niveau* svarer til sprogets implementation. I udgangspunktet er dette usynligt for brugeren af sproget (programmøren), og i mange tilfælde irrelevant og måske endog uinteressant. I forrige forelæsning udviste vi dog interesse for dette niveau af CLOS, idet vi studerede en mulig, principiel implementation af sproget (som kunne udvikle sig til en legetøjsimplementation). Derfor kan vi genkende en del af de primitiver, som er afbildet over det interne niveau (som vist ovenfor).

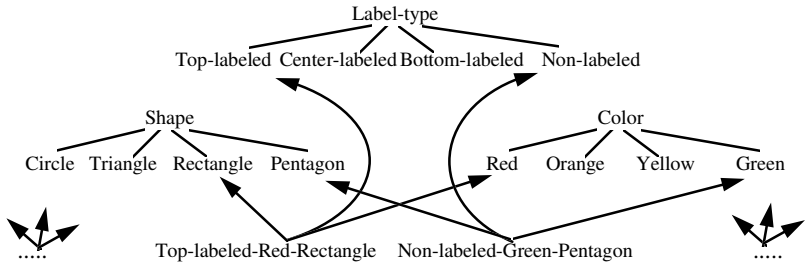
MOP niveauet (metaprotokol niveauet), som er vist forneden mellem det eksterne og interne niveau er en omhyggelig udvalgt delmængde af implementationen, som man har offentliggjort og ikke mindst dokumenteret for brugere af sproget. Ved at definere klasser (subklasse af eksisterende metaklasser), og ved at definere nye metoder i generiske funktioner kan man flytte sproget rundt i design rummet. Bemærk, at omfanget og naturen af metaobjekt protokollen afgør "størrelsen" af designrummet.

Programmatisk skabelse af klasser (1).

Problemet: Multipel nedarvning og brug af 'mixin classes' muliggør kombinatorisk definition af et meget stort antal klasser, hvoraf måske kun en mindre delmængde anvendes i et konkret program.

En løsning: Klasser defineres efter behov, i takt med at vi har brug for at instantiere klasserne.

Pointe: Når klasser er objekter (instanser af metaklasser) kan oprettelsen af klasser underlægges programmatisk kontrol.



PS4 - Metaobjekt Protokoller

Noter

Vi viser eksempel et på tre klassehierarkier, hvoraf selve den geometriske form er den substantielle, og hvor farven og label typen er mixins.

Via multipel nedarvning kan vi alene af de viste klasser danne $4 \cdot 4 \cdot 3 = 48$ forskellige "aggregate classes". I mange applikationssammenhænge har vi kun brug for instanser af en lille delmængde af disse. Derfor vil det være spild af plads (og spild af programmøranstrengelser) at skulle definere alle disse kombinationer.

Hvad mener vi egentlig med 'programmatisk' i overskriften, og hvad er modsætningen? Normalt skabes klasser implicit, som følge af evaluering af en (defclass ...) form. Klasse definition udøves normalt ikke under kontrol af et program; det er derimod brugeren der direkte affyrer klassedefinitioner, og dermed afgør brugeren med dette, hvilke klasse metaobjekter der skal laves. I det ovenstående har vi gjort oprettelsen af disse klasse metaobjekter til en del af programmet. I takt med at programmet afvikles vil der kunne oprettes nye klasser (klasse metaobjekter). Vi siger at oprettelsen af klasser er blevet et programmatisk element.

Programmatisk skabelse af klasser (2).

Eksempel på anvendelse:

```
(make-programmatic-instance '(Rectangle Red Top-labeled) :title "PS4"
                                                         :side1 5 :side2 6)
```

Definitioner:

```
(defun make-programmatic-instance (superclass-names &rest initargs)
  (apply #'make-instance
         (find-programmatic-class
           (mapcar #'find-class superclass-names))
         initargs))

(defun find-programmatic-class (superclasses)
  (let ((class  en evt. allerede eksisterende metaobjekt for klassen)
    (if class
        class
        (make-programmatic-class superclasses))))

(defun make-programmatic-class (superclasses)
  (make-instance 'standard-class
    :name (mapcar #'class-name superclasses)
    :direct-superclasses superclasses
    :direct-slots ()))
```


Noter

En måde at finde ud af, om vi allerede har lavet klassen i funktionen find-programmatic-class er følgende:

1. Opsøg alle subklasser af f.eks. første klasse i superclasses.

2. Hvis én af disse subklasser netop har superklasserne 'superclasses' antager vi, at klasse eksisterer, og vi kan blot returnere denne.

Læg mærke til, at vi i make-programmatic-instance navngiver den nye klasse med en liste; i eksemplet '(Rectangle Red Top-labeled). Navngivning er ikke et væsentligt emne i dette eksempel. Vi finder frem til en evt. tidligere programmatisk defineret klasse via dets superklasser (jf. find-programmatic-class), og *ikke* via klassens navn.

Program observerende protokoller.

- Ved at *offentliggøre* og *dokumentere* dele af CLOS implementationen muliggøres observation og analyse af CLOS programmer fra CLOS programmer.
- Relaterede dele af implementationen samles i en *protokol.*

- Definitioner i *introspection protokoller* for klasser, generiske funktioner, og metoder kan anvendes
  - som grundlag for implementation af program browsere.
    - Eks.: super- og subklasselister
  - til specialiserede forespørgsler om programegenskaber.
    - Eks.: Find alle relevante generiske funktioner på en klasse, undtagen accessorer.

Dele af introspection protokol for klasser:

(class-name <*class*>) Returner klassens navn.

(class-direct-super-classes <*class*>) Returnerer direkte superklasser af <*class*>.

(class-direct-slots <class>) Returner direkte slots af <*class*>.

(class-precedence-list <*class*>) ...

(class-slots <*class*>) ...

(class-direct-subclasses <*class*>) ...

(class-direct-methods <*class*>) ...

Noter

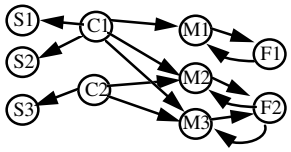
På engelsk, og i bogen *The art of the metaobject protocol* benyttes termen introspection om ovenstående. 'Introspection' betyder "observation og analyse af sig selv" [oversat fra Webster's ordbog].

Kapitel 2 af bogen beskriver i relativ stor detaljer introspection. Kapitlet er relativ let læst, idet stoffet er elementært. Det mest spændende eksempel fra kapitlet beskriver vi kort på næste side.

Vi genkender de generiske funktioner i introspection protokollen for klasser (vist nederst til højre ovenfor) som de accessorer, vi definerede på standard-class i forrige kapitel.

Program-skabende protokoller.

- Et CLOS program repræsenteres som et netværk af metaobjekter.
- Det er muligt for ét CLOS program at skabe et andet CLOS program.
  - Et sådant CLOS program vil være at betragte som et programudviklingsværktøj (en slags editor).
  - Funktionerne ensure-class (og tilsvarende for metoder og generiske funktioner) er de relevante primitiver.
- Det er muligt af rekontruere overflade formerne [(defclass ...) mv.] fra netværket af metaobjekter.
  - Netværket af metaobjekter: *intern program repræsetation*.
  - Overflade formerne: *ekstern program præsentation*.
- Netværket af metaobjekter kan opfattes som den primære, interne programrepræsentation i en *editor iboende programmeringsomgivelse* (*residential programming environment*).



PS4 - Metaobjekt Protokoller

Noter

Nederst viser vi et eksempel på et netværk med klasser (C), metoder (M), generiske funktioner (F) og slots (S).

Det er muligt at lave struktur-orienteret editorværktøj af forskellig art, der hjælper med til at konstruere sådanne netværk. Jeg har selv arbejdet med dette, og dokumenteret det i artiklen "A Hyperstructure Programming Environment for CLOS" i TOOLS4, Technology of Object-Oriented Languages and Systems (editor Jean Bézivin og Betrand Meyer), Prentice Hall. (Artiklen er også trykt som teknisk rapport IR 91-03 fra Afdelingen for Matematik og Datalogi ved AUC).

Det er relevant at sammenligne diskusionen på denne side med de såkaldte uniforme modeller af programmeringsomgivelser i kapitlet Programmeringsomgivelser til Lisp.

Sprog-udvidende protokoller.

- **Situation**: Implementationen af CLOS (fortolkeren) manifesterer sig som generiske funktioner, hvis metoder specialiserer på metaklasserne.
- **Problemet**: Hvordan ændres semantikken af sproget for udvalgte CLOS programmer?
- **Løsning**:
  - Definer en subklasse af en eksisterende metaklasse.
  - Definer nye metoder i eksisterende generiske funktioner på den nye subklasse.
  - Arranger metaklasse-instantiering således, at den nye subklasse bliver instantierret i stedet for standard metaklassen.

Noter

Protokollerne som beskrives her og på de følgende sider, kaldes i bogen *The Art of the Metaobject Protocoll* for intercessory protocols. 'Intercessory' er afledt af 'intercede', som bl.a. betyder at mægle. Man kan diskutere om dette er et godt ord at tage i anvendelse i denne sammenhæng. Jeg foretrækker derfor betegnelsen 'sprog-udvidende protokoller'. Disse protokoller beskrives bl.a. i kapitel 3 af bogen.

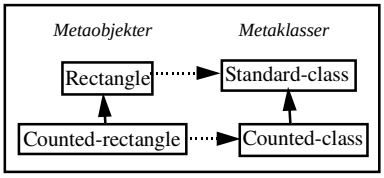
Et simpelt eksempel.

Vi ønsker at nærmere bestemte klasser (instanser af counted-class) skal kunne tælle dereres instanser.

```
(defclass Counted-class (standard-class)
  ((counter  :initform 0)))

(defmethod make-instance :after
                ((class Counted-class) &key)
  (setf (slot-value class 'counter)
        (+ 1 (slot-value 'counter))))

(setf (find-class 'Counted-rectangle)
  (make-instance 'Counted-class
     :name 'Counted-rectangle
     :direct-superclasses
         (list (find-class 'Rectangle))
     :direct-slots ()))
```




```
(defclass Counted-rectangle (Rectangle)
  ()
 (:metaclass Counted-class))
```

PS4 - Metaobjekt Protokoller

Noter

Nederst vises, hvordan klassen Counted-rectangle kan defineres. Et Counted-rectangle er en instans af den nye metaklasse Counted-class, og et Counted-rectangle har som sådan et felt, som hedder counter. Dette felt holder styr på, hvormange instanser af Counted-rectangle der er lavet.

Det ses, at definitionen er relativt omstændelig, og vi er stærkt motiveret for at finde en kortere og mere direkte udtryksform. Nederst til højre vises et eksempel på en anvendelse af defclass, som tager en ny klasseoption, nemlig metaclass. Denne klasseoption er netop beregnet til at angive metaklassen af den klasse, som vi er igang med at definere. Det er nu op til makroen defclass og ensure-class (som vi så på i forrige forelæsning) at samle denne information op, og at anvende informationen til at styre klassen af metaobjektet, der repræsenterer Counted-rectangle.

After metoden i make-instance funktionen (som givet ovenfor) specialiserer på klassen Counted-class. Dette er OK idet instantiering foregår ved at kalde funktioner på metaobjekterne. De relevante metoder findes derfor i klasserne af metaobjekterne, altså metaklasserne. Resultatet af make-instance (primær) metoden er et objekt der er en instans af klassen, som er repræsenteret ved metaobjektet.

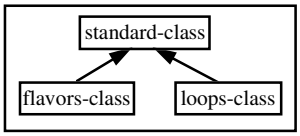
Problem: Hvis vi ønsker at overholde invarianten

> "Tælleren i Counted-class angiver hvormange objekter der på ethvert tidspunkt eksisterer af instanser af Counted-class (som jo er klasser, ala Counted-rectangle)."

er den foreslåede after metode, som givet ovenfor, ikke tilstrækkeligt. Problemet er naturligvis, at objekter af typen Counted-class også kan dø. Hvilke løsninger kan man foreslå, således at invarianten holder?

Kontrol over class precedence listen.

- Givet beslutningen om en lineær ordning af alle superklasserne til en klasse bestemmer class-precedence listen den indbyrdes dominans af metoder og slots.
- Der findes findes forskellige variationer af objekt-systemer (Loops, Flavors og CLOS) til Lisp der bl.a. adskiller sig ved forskellig definition af class-precedence listen.
- Flavors og Loops klasser og -objeker kan sameksistere med CLOS klasser og objekter i et kørende CLOS system.



- Den generiske funktion compute-class-precedence-list dokumenteres som en del af metaobjekt protokollen:
- (compute-class-precedence-list *class*).
  - En permuation af af superklasse metaobjekter af *class*, hvoraf den første er *class* selv, og de to sidste er standard-object og t.
  - Class precedence listen af *class* må ikke muteres efter den er skabt.
  - Class precedence listen beregnes én gang af den generiske funktion compute-class-precedence-list, og returneres herefter ved kald af den generiske funktion class-precedence-list (acessor på klasse metaobj.)

Noter

I lighed med det forrige eksempel opnår man muligheden for sameksisterende klasser og objekter fra forskellige varationer af Objekt-systemet ved at definere primærmetoden compute-class-precedence-list på hhv. flavors-class og loops-class.

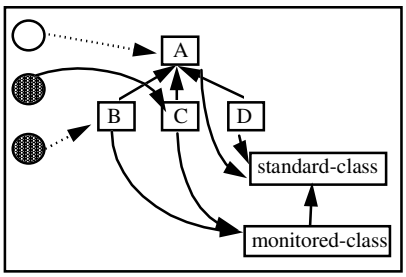
Kontrol af slot access.

- Slot access foregår normalt gennem generiske access funtioner.
  - Tillader kontrol af slot access pr. slot i alle instanser af en klasse.
- På et lavere niveau foregår slotaccess gennem den generiske funktion slot-value og (setf slot-value).
  - Tillader kontrol af slot access pr. instans af alle klasser, med en speciel metaklasse (såsom monitored-class).

```
(defun slot-value (instance slot-name)
 (slot-value-using-class
   (class-of instance) instance slot-name)))

(defmethod slot-value-using-class
      ((class standard-class)  instance slot-name)
 ...)

(defmethod slot-value-using-class :before
     (class monitored-class) instance slot-name)
 (note-operation instance slot-name 'slot-value))
```



Noter

Funktionen note-operation registrer blot på en eller anden måde det tilfælde af slot-access, som er forekommet (putter informationen på en liste, eller lignende).

Hvorfor er det nødvendigt med den ekstra parameter af slot-value-using-class i forhold til slot-value? Vi ønsker at diskriminere på typen af klassen, og ikke kun typen af applikationsobjektet (de runde ovenfor). Derfor kan vi opnå samme opførsel for alle instanser af en given klasse. Eksempelvis vil alle instanser af klasserne B og C have overvågede slots.

Det er værd at bemærke, at vi ved brug af multipel nedarvning kan lave nye metaklasser, som kombinerer variationerne i sprogets semantik. Eksempelvis kan vi lave en ny metaklasse, som både arver fra monitored-class ovenfor og fra counted-class, som vi så på tidligere. På denne måde kan vi elegant kombinere de semantiske variationer, vi har befordret ved at lave metoder på specialiserede metaklasser.