11 Implementation af CLOS.

Begreber og problemer.
Lisp i Lisp via metacirkulær fortolkning.
CLOS implementationsmodeller.
CLOSETTE.
Metaklassehierarkiet.
Repræsentation af klasser.
Repræsentation af generiske funktioner.
Repræsentation af metoder.
Andre metaobjekter.
Kald af generisk funktion.
Bootstrapping problemer.

PS4 - Implementation af CLOS

Noter

Forelæsningen er primært baseret på kapitel 1 i bogen *The art of the the Metaobject Protocol* af Gregor Kiczales et al. Bogen er fra MIT press, og fra 1991. Jeg har for et par år siden anmeldt denne bog. Her er hvad jeg skrev dengang:

The Common Lisp Object System (CLOS) is an object-oriented language on top of Common Lisp. This book is about a metaobject protocol for CLOS. The CLOS metaobjects are the objects that represent classes, methods, generic functions, and some other aspects of a program. The metaobject protocol is a well-documented functional interface to the metaobjects, which allows programmers to tailor and specialize the CLOS language, e.g., in order to make it more similar to another object-oriented language, or to gain better efficiency with respect to some specific mechanism.

The book consists of two main parts: (1) A tutorial part, in which the reader is guided through a relatively simple implementation of Closette (a subset of CLOS) followed by development and applications of a metaobject protocol for Closette; And (2) a reference part, which is the detailed specification of a metaobject protocol for full CLOS. The first part of the book is centered around a number of sample extensions to Closette. Besides these two parts, the book contains a short introduction to CLOS itself and a full source listing of the implementation of Closette.

Although the topic of this book should be of general interest to people involved with programming language and programming environment design, the book is written in a style, which makes it difficult for people outside the Lisp-culture to follow the discussions. Readers of the book need to be familiar with both object-oriented concepts, Common Lisp, and CLOS. However, for people with this background, the book is very interesting and profitable. Ever since the publication of the CLOS specification back in 1988 it has been a frustrating experience to understand and work with the existing fragments of the metaobject protocol. This book both explains the philosophy and the design considerations behind the protocol, and it contains the specification of the current state of the protocol.

(fortsættes næste noteark).

Begreber.

- *Metasprog*: Et sprog som bruges til at tale om et andet sprog. [Oversat fra Webster].
- *Metalinguistisk abstraktion*: etablering af nye sprog i termer af eksisterende sprog.
  - Via en *fortolker*.
    - Det nye sprog kaldes et *indlejret sprog* (embedded language) i forhold til implementationssproget.
    - *Metacirkulær fortolker*: En fortolker, der er skrevet i det samme sprog, som det fortolker.
- En *metaobjekt-protokol* er et interface til et sprog, som gør det muligt for en programmør at tilgå og modificere et objekt-orienteret sprogs egenskaber og implementation.
  - En metaobjekt protokol udgøres af metoder på metaklasserne, og som derfor virker på metaobjekterne.
  - Metaobjekt-protokoller er emnet i næste forelæsning.

Noter

Om bogen "The art of the Metaobject Protocol", fortsat:

The general topic of this book can be understood as the art of making open languages without requiring the user to understand the full language implementation. The languages need not to be object-oriented, but in order to take advantage of the techniques described in the book the language implementations must be based on object-oriented concepts. Even more general, the topic may be phrased as he art of programming open applications in object-oriented languages. It would have been interesting if the book had contained stronger discussions on the more general aspects of designing and programming such systems. Although the book contains scattered elements of general advice and experience on the topics, the impression remains that the book is narrowly focussed on one particular metaobject protocol; the one for CLOS.

It is a rewarding experience to read and be able to understand non-trivial programs. Needless to say, the programming and documentation styles have to be carefully adjusted to make such reading and understanding possible. Some times the words ``literate programming" are being used for such styles. Considered as a literate program, the parts of the book, which describe Closette and its metaobject protocol, are certainly worthwhile. However, a more systematic approach to literate programming would be necessary if more aspects of the language implementation and the metaobject protocol should be exposed. It would be a real challenge, and probably a rewarding one, to read a literate program with the full CLOS implementation. This is not part of this book, but ``The art of the Metaobject Protocol" is a good beginning.

Metacirkulær fortolker: Lisp i Lisp.

```
(define (eval exp env)
  (cond ((self-evaluating? exp) exp)
        ((quoted? exp) (text-of-quotation exp))
        ((variable? exp) (lookup-variable-value exp env))
        ((definition? exp) (eval-definition exp env))
        ((assignment? exp) (eval-assignment exp env))
        ((lambda? exp) (make-procedure exp env))
        ((conditional? exp) (eval-cond (clauses exp) env))
        ((application? exp)  (apply (eval (operator exp) env)
                                    (list-of-values (operands exp) env)))
        (else (error "Unknown expression type -- EVAL" exp))))

(define (apply procedure arguments)
  (cond ((primitive-procedure? procedure
           (apply-primitive-procedure procedure arguments))
        ((compound-procedure? procedure)
           (eval-sequence (procedure-body procedure)
                          (extend-environment
                             (parameters procedure)
                             arguments
                             (procedure-environment procedure))))
        (else  (error "Unknown procedure type -- APPLY" procedure))))
```

PS4 - Implementation af CLOS

Noter

Konkret vises ovenfor en Scheme implementation af (starten på) en metacirkulær fortolker for Scheme.

Eksemplet er taget fra bogen *Structure and Interpretation of Computer Programs.* Alt i alt fylder eksemplet 325 linier, men så er Scheme fortolkeren også programmeret pænt med et hav af selektorer, der fremmer dataabstraktions princippet. Detaljerne omkring environments fylder ca. 1/3 af dette linieantal. Hvis man udtrykker sig kortets muligt, uden brug af en række små funktioner og uden information hiding, kan man mageligt skrive en fuld Lisp fortolker på et A4 ark! Dette har vi allerede set et eksempel på i artiklen “Programming with Continuations”. John McCarthy (ophavsmanden til Lisp) illustrerer også dette i artiklen “A Micro-Manual for Lisp”, Sigplan Notices, vol 13, no. 8, 1978 (Proceedings fra den 1. konference om programmeringssprog historie).

Vi viser kun eval og apply, som er de centrale top-niveau funktioner. Eval er altså den centrale evalueringsfunktion i implementationssproget. I mange Lisp systemer (dog ikke i Scheme!) stilles eval tilrådighed i det nye sprog. Dette er en form for metafacilitet: sprogimplementations primitivet stilles til rådighed gennem selve sproget.

I alle Lisp systmer (også Scheme) findes endvidere også apply. Dette er primitivet, som tillader os at aktivere en procedure på en liste af argumenter (som man har i hånden). Altså

```
(apply f lst)
```

hvor lst kunne være (list a b c) frem for

```
(f (car lst) (cadr lst) (caddr lst))
```

Interesserede kan konsultere hele implementation af Scheme i Scheme på filen /user/normark/public/ps4/scheme-in-scheme.

Oversigt over CLOS implementationsmodeller.

- '**Native CLOS**'.
  - CLOS implementation, som udnytter viden om et specifik Lisp systems implementationsdetaljer.
- '**Portable CLOS**':
  - CLOS implementation som er implementeret (i størst muligt omfang) i Common Lisp.
  - PCL (Portable Common Loops).
- **Legetøjs implementationer**.
  - Implementeret helt i Common Lisp.
    - CLOSETTE: En kørende implementation af en delmængde af CLOS.
      - Alvorlige effektivitetsproblemer.
  - Metacirkulær fortolker for CLOS i CLOS.
- **Principielle implementationer**.
  - Implementeret i CLOS
    - CLOSETTE i CLOS
      - Kapitel 1 i bogen *The art of the Metaobject Protocol.*
      - En ikke kørende implementation.
  - Beregnet som *pædagogisk fremstilling* af 'objekt-orienteret implementation af et objekt-orienteret sprog med metaklasser'

PS4 - Implementation af CLOS

Noter

Den version af CLOS, som kører i CMU Common Lisp, er PCL. PCL er udviklet på Xerox, og fik på et tidligt tidspunkt derfor navnet Portable Common Loops (fordi det var her CLOS forgængeren Loops blev udviklet).

Når vi siger legetøjsimplementation mener vi ikke "det rene pjat". Leg er for børn udviklende og udforskende; således også for sprogdesignere og implementationsfolk, som hurtigt kan få nogle første, vigtige erfaring med sproglige ideer ved at "lege med disse" i en metacirkulær fortolker. Eksempelvis er det utrolig let at ændre den viste metacirkulære fortolker for Scheme fra applikativ order til normal order evaluation (og det var jo nok værd at indhøste nogle første konkrete erfaringer med, hvis man var den første, der havde fået den ide!)

CLOSETTE i CLOS.

- CLOSETTE er en legetøjsimplementation af CLOS.
- Under antagelse af, at CLOS eksisterer, vil vi
  - studere en objekt-orienteret repræsentation af klasser, generiske funktioner og metoder (og andre sprogelementer) i CLOSETTE.
    - klasser, metoder og generiske funktioner er metaobjekter, som er
    - instanser af metaklasser.
  - studere en fortolker for Closette, der er organiseret som metoder på metaklasserne.
- I en egentlig implementation af Closette bootstrappes direkte fra et lavere niveau end CLOS: fra Common Lisp.
  - Vi studerer således *ikke* en rigtig metacirkulær fortolker for CLOS.
  - Vi ser derimod på en *principiel implementation* af CLOSETTE, som indfanger de essentielle ideer omkring
    - *objekt-orienteret repræsentation* af et objekt-orienteret program.
    - *refleksion* i termer af veldefinerede protokoller af metoder i metaklasserne: *metaobjekt protokoller.*

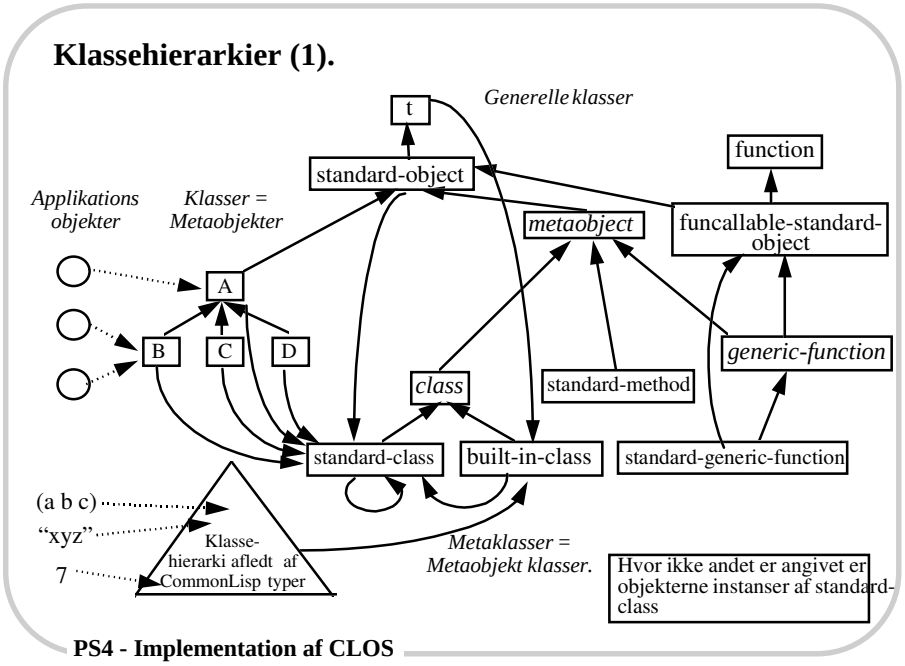
Noter

I første omgang vil vi se på en principiel implementation af CLOS delmængden, som kaldes CLOSETTE. Dette kan betegnes som situationen **bag ved scenen**, som vi her blotlægger (i en ren og pæn form; virkeligheden er givetvis mere broget). Dette giver os indsigt i de basale dele, og det gør os bekendt med de klasser, hvorpå vi senere vil studere metaobjekt protokoller.

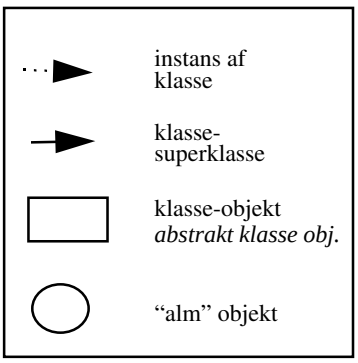
Metaobjekt protokoller hører til det egentlig interessante, som vi vil diskutere. Man kan sige at blotlægningen af CLOS implementationen lader op til (og tillader os at forstå bedre) de metaobjekt protokoller, som vi senere vender tilbage til. Metaobjekt protokoller kan siges at befinde sig på **grænsen mellem forscenen og bagscenen.** Vi helliger næste forelæsning til dette emne.

På **forscenen** befinder vore egne objekt-orienterede CLOS programmer sig.

Læs mere i bogen om denne analogi.



Noter



Hosstående ramme viser en symbolforklaring af ovenstående. Vi benytter de samme symboler, som vi tidligere så i forbindelse med studiet af metaklasser i Smalltalk.

Som sædvanlig er klasserne A, B, C og D eksempler på klasser, som formodes at være defineret af en "almindelig programmør".

I CLOS bruges betegnelsen 'metaobjekt' som en fællesbenævnelse for alle instanser af (subklasser af) den abstrakte klasse, som kort og godt hedder *metaobject*.

I klassehierarkiet ovenfor har vi ikke vist alle klasserne. Vi har undladt nogle abstrakte klasser, samt en del metaklasser: standard-accessor-method, slot-definition, direct-slot-definition, specializer mv. Hierarkiet ovenfor udgør dog det essentielle.

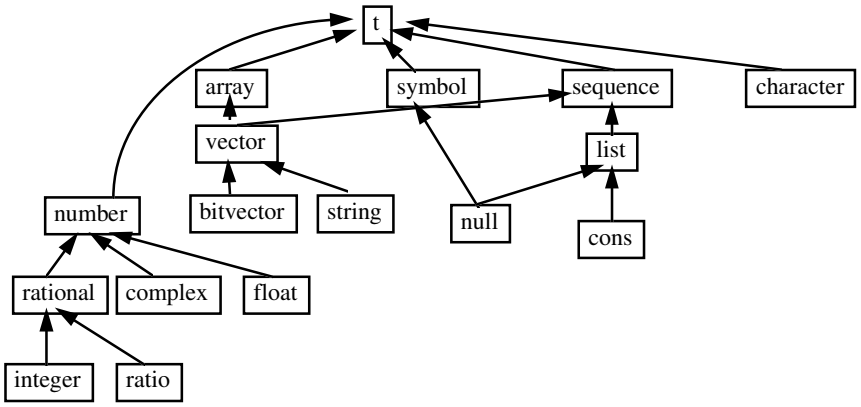
Kursiverede klasseobjekter er abstrakte (der er ingen instanser af disse).

Det i trekanten antydede hierarki er alle klasser, der er instanser af built-in-class. Alle disse klasser er subklasser af t. (Denne relation er ikke vist på tegningen, for ikke af forkludre figuren for meget).

Modellen ovenfor er væsentlig simplere end den tilsvarende Smalltalk model, som vi studerede i et tidligere kapitel (sammenlign med siden "Afrunding af klassehierarki og metaklassehierarki"). Hovedbevæggrunden til det mere indviklede hierarki i Smalltalk var øjensynlig ønsket om at have nuancerede instantierings/initialiserings metoder i metaklasserner (new, og hvad man måtte finde på af alternative betegnelser); derfor det parallelle metaklassehierarki i Smalltalk. **Hvorfor har vi ikke brug for noget lignende i CLOS?** Årsagen er nok den, at CLOS allerede har mekanismer (initform, initarg mv.) i klassebeskrivelsen, der tager hånd om initialisering af slots fra et kald af (make-instance ...). Så der er ikke det store behov for at lave specialiserede udgaver af make-instance i subklasser af standard-class.

Klassehierarkier (2).

Hierarkiet af klasser afledt fra Common Lisp typer.



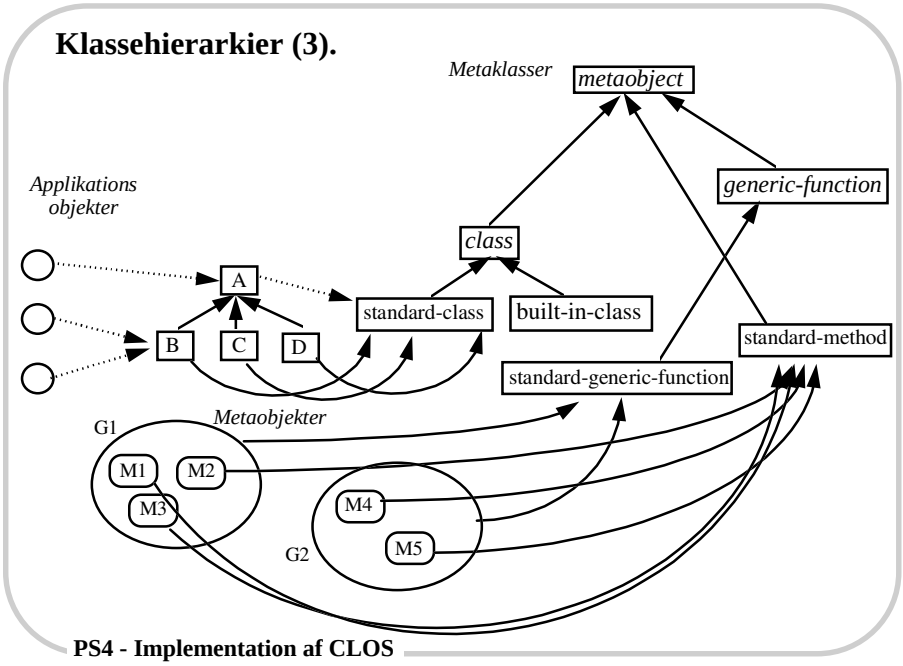
Noter

Dette svarer til hierarkiet, som er antydet på den tidligere viste figur med klassehierarkierne for CLOS. Alle de viste klasser (metaobjekter) vil typisk (men dog ikke nødvendigvis) være instanser af built-in-class.

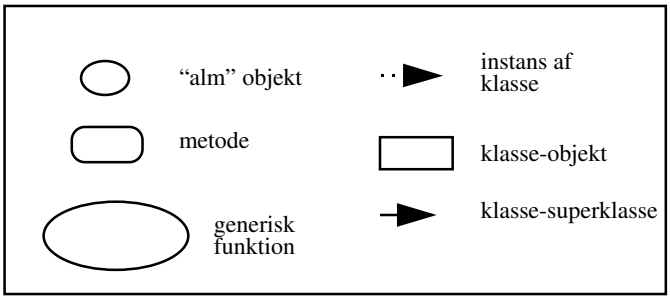
Observationer og principper.

- Instanser af **standard-class** er klasser, som beskriver tilstand ved brug af slots.
  - Et objekt, der har standard-class som metaklasse, har nul eller flere slots.
- **standard-object** er superklasse til alle klasser, der er instanser af standard-class. (Undtagen sig selv: standard-object er naturligvis ikke sin egen superklasse).
  - Standard-object er superklasse til alle klasser, der beskriver tilstand ved brug af slots.
- Instanser af **built-in-class** er klasser, der er afledt af Common Lisp typer.
  - Et objekt, der har built-in-class som metaklasse, har en speciel (og for CLOS ukendt) repræsentation.
  - Begrænsninger:
    - Instanser af built-in-class kan ikke subclasses
    - Instanser af built-in-class kan ikke instantieres ved make-instance.
    - Tilstanden af objekter, der har built-in-class som metaklasse, kan ikke tilgåes med slot-value.
  - Man kan specialisere metoder på instanser af built-in-class.

Noter



Noter



På denne figur illustreres ikke blot "almindelige" klasser og deres instanser, men også metaobjekterne, der repræsenterer metoder og generiske funktioner. Man ser, at disse objekter hhv. er instanser af klasserne standard-method og standard-generic-function.

Bemærk, at vi ikke har vist den øvre del af klassehierarkiet på denne figur. Vi koncentrerer os om de "nye" metaobjekter.

Repræsentation af klasser.

En klasse repræsenteres af en instans af klassen standard-class.

```
(defclass standard-class ()
     ((name :initarg :name :accessor class-name)
      (direct-superclasses  :accessor class-direct-superclasses)
      (direct-slots :accessor  class-direct-slots)
      (class-precedence-list :accessor class-precedence-list)
      (effective-slots :accessor class-slots)
      (direct-subclasses :initform () :accessor class-direct-subclasses)
      (direct-methods :initform () :accessor class-direct-methods)))
```

```
(defmacro defclass (name direct-superclasses direct-slots
                    &rest options)
  `(ensure-class ',name
    :direct-superclasses
     ,(canonicalize-direct-superclasses direct-superclasses)
    :direct-slots
     ,(canonicalize-direct-slots direct-slots)
    ,@(canonicalize-defclass-options options)))
```

```
(defun ensure-class (name &rest all-keys)
  (if (find-class name nil)
      (error "Can't redefine class")
      (let ((class (apply
                    #'make-instance
                    'standard-class
                    :name name all-keys)))
        (setf (find-class name) class)
        class)))
```

PS4 - Implementation af CLOS

Noter

Øverst vises en definition af klassen standard-class. Klasser repræsenteres som instanser af denne klasse. Nederst til venstre vises en Common Lisp makro definition. En makro i Lisp er en syntaktisk abstraktion. Makroen ovenfor implementerer overfladesyntaksen af den bekendte defclass form. Denne oversættes til et kald af ensure-class, der er en almindelig Common Lisp funktion (vist nederst til højre).

Canonicalize-... funktionerne transformerer informationen, som givet i defclass formen, til en intern form, som er mere hensigtsmæssig at arbejde med. Superklasserne konverteres til en liste af referencer til andre instanser af standard-class. Direkte slots kanonikaliseres til en 'property list' af egenskaber. (En propertyp liste er en liste på formen (egenskab1 værdi1 egenskab2 værdi2 ...), altså altid en liste med et lige antal elementer).

Funktionen find-class, samt mutator-funktionen (setf find-class), bestyrer afbildningen fra klassenavne til metaobjekter, der repræsenterer klasser. Se disses definition i bogen.

Der defineres ud over de her viste funktioner en **after-metode i den generiske funktion initialize-instance**, som udfører diverse initialiseringer af klasseobjektet:

- default-værdi af superklasser,
- links til direkte subklasser,
- konvertering af slot property lister til instanser af slotdefinitions metaobjekter.

Metoden definerer også **accessor metoder på klassen**, ved brug af ensure-method (se senere). Den sidste handling i denne metode er et kald af funktionen finalize-inheritance, som vi ser på på næste side.

Der er **tre niveauer** af i behandlingen af metaobjekterne:

1. Et syntaktisk niveau, afspejlet af makrodefinitionerne.
2. Et sammenbindings niveau, som mapper navne til metaobjekter.
3. Et semantisk niveau, som helt og holdent arbejder på metaobjekterne, og hvor deres opførsel bestemmes af metoder på metaklasserne.

Disse tre niveauer genfinder vi mange andre steder herefter. Understregede navne herover afspejler, at dette program element er vist andetsteds i materialet.

Nedarvning og 'class precedence list'.

Kaldes af after metoden initialize-instance, når denne anvendes på en standard-class

```
(defun finalize-inheritance (class)
  (setf (class-precedence-list class)
        (compute-class-precedence-list class))
  (setf (class-slots class)
        (compute-slots class))
  (values))


(defun compute-class-precedence-list (class)
  (let ((classes-to-order (collect-superclasses* class)))
    (topological-sort classes-to-order
                      (remove-duplicates
                          (mapappend #'local-precedence-ordering
                                     classes-to-order))
                      #'std-tie-breaker-rule)))

(defun collect-superclasses* (class)
  (remove-duplicates
    (cons class (mapappend #'collect-superclasses*
                           (class-direct-superclasses class)))))
```

PS4 - Implementation af CLOS

Noter

På denne slide viser vi finalize-inheritance, og hvorledes class precedencelisten beregnes. **Collect-superclasses*** opsamler alle superklasserne (som metaobjekter) af den givne klasse. Idet klasser kan nåes ad mere end én sti fjernes duplikater.

(values) betyder 'ingen værdi'; teknisk set er funktion value et primitiv, der kan returnere en multipel værdi uden at skulle aggregere med eksempelvis lister; ingen værdi er et specialtilfælde af en multipel værdi.

Topologisk sortering (ikke vist) kaldes med tre parametre: klasserne der skal ordnes (og som er resultatet af ovenstående opsamling), en liste af par, der beskriver de primitive bidrag til ordningen af klasser (jf. slide "Class Precedence Listen" i kapitlet Introduktion til CLOS), og en funktion, der løser op for konflikter (når to eller flere class precedence lists kan være resultatet, vælges netop én af disse; reglen blev givet på ovennævnte slide's noter; funktionen std-tie-breaker-rule implementerer denne udvælgelse).

Vi viser ikke på denne slide definitionen af funktionen **compute-slots**. Formålet med denne er at beregne en liste af alle slots af den pågældende klasse, incl. de nedarvede. Resultatet er en liste af *effective slot definition metaobjects.*

Instantiering og initialisering.

```
(defgeneric make-instance (class &key))

(defmethod make-instance ((class symbol) &rest initargs)
  (apply #'make-instance (find-class class) initargs))

(defmethod make-instance ((class standard-class) &rest initargs)
  (let ((instance (allocate-instance class)))
    (apply #'initialize-instance instance initargs)
    instance))
```

Initialize-instance initialiserer slots ud fra initialiseringsargumenter og initforms mv.

```
(defun allocate-instance (class)
  (allocate-std-instance
    class
    (allocate-slot-storage (count-if #'instance-slot-p (class-slots class))
                           secret-unbound-value)))
```


Noter

Øverst viser vi to metoder i den generiske funktion make-instance. Den første af disse implementerer tilfældet, hvor make-instance gives et symbol. Symbolet slåes op med find-class, hvorefter den anden af metoderne kaldes.

Som det ses, involverer instantiering allokering af lager, og initialisering. Dette er også, hvad vi ville forvente.

Vi viser allocate-instance, som forestår selve lagerallokeringen til en instans af den givne klasse. allocate-instance tager som 1. parameter klassen, som der skal allokeres plads til, og som 2. parameter den plads, der er behov for.

Primitivet **allocate-slot-storage** allokerer den fysiske plads til slots; størrelsen af denne plads (1. parameter) afhænger naturligvis af, hvor mange slots der er instans-allokerede (afgøres af instance-slot-p). Secret-unbound-value (2. parameter) er en variabel, der indeholde værdien, der opfattes som "unitialiseret". Denne værdi puttes initielt ind i de enkelte felter i objektet.

Repræsentation af generiske funktioner.

En generisk funktion repræsenteres af en instans af klassen standard-generic-function.

```
(defclass standard-generic-function ()
    ((name :initarg :name :accessor generic-function-name)
     (lambda-list  :initarg :lambda-list :accessor generic-function-lambda-list)
     (methods :initform () :accessor generic-function-methods)))
```

```
(defmacro defgeneric (function-name lambda-list &rest options)
  `(ensure-generic-function
    ',function-name
    :lambda-list ',lambda-list
    ,@(canonicalize-defgeneric-options options)))

(defun ensure-generic-function  (function-name &rest all-keys)
  (if (find-generic-function function-name nil)
      (find-generic-function function-name)
      (let ((gf (apply #'make-instance 'standard-generic-function
                       :name function-name
                       all-keys)))
        (setf (find-generic-function function-name) gf)
        gf)))
```

PS4 - Implementation af CLOS

Noter

Ovenfor viser vi metaklassen for generiske funktioner, som kaldes standard-generic-function.

Nedenfor viser vi makroen, som formidler overfladesyntaksen for defgeneric i CLOS. Vi ser, at ligesom for klasser kaldes en ensure-funktion, her ensure-generic-function. Lambda-listen er, som bekendt, parameterlisten af den generiske funktion.

I ensure-generic-function returneres en evt. eksisterende generisk funktionsobjekt under navnet function-name. Hvis en sådan ikke eksisterer oprettes en ny instans af klassen standard-generic-function. Denne registreres, og objektet returneres.

Funktionen find-generic-function søger efter den givne funktion; hvis 2. parameter er nil, vil den aldrig give en fejl, hvis funktionen ikke findes.

Repræsentation af metoder (1).

En metode repræsenteres af en instans af klassen standard-method.

```
(defclass standard-method ()
  ((lambda-list :initarg :lambda-list :accessor method-lambda-list)
   (qualifiers :initarg :qualifiers  :accessor method-qualifiers)
   (specializers :initarg :specializers  :accessor method-specializers)
   (body :initarg :body :accessor method-body)
   (environment :initarg :environment  :accessor method-environment)
   (generic-function :initform nil  :accessor method-generic-function)))
```

```
(defmacro defmethod (&rest args)
 (multiple-value-bind (function-name qualifiers
                       lambda-list specializers body)
     (parse-defmethod args)
  `(ensure-method (find-generic-function ',function-name)
    :lambda-list ',lambda-list
    :qualifiers ',qualifiers
    :specializers ,(canonicalize-specializers specializers)
    :body ',body
    :environment definition-environment )))
```


Noter

Repræsentationen af metoder følger mønsteret, som vi har set for klasser og generiske funktioner. Qualifiers er rollen af metoden. Specializers er listen af specialiseringer (klasser eller (eql objekt) specialiseringer). Environment er den omgivelse, hvori metoden er defineret. Generic-function henviser naturligvis til den generiske funktion, som indeholder metoden.

Øverst ses klassen, hvoraf hver metode er en instans.

Nederst ses makroen, som forsyner os med overflade syntaks for metode definition. Som tidligere ekspanderer denne makro blot til en ensure-... funktionskald. Læg mærke til kaldet af **parse-defmetod** (hvis definition ikke vises her). Formålet med dette kald er at splitte metode-definitionen i dens bestanddele.

Definition-environment er den omgivelse, hvori metoden er defineret. Dette er som regel top-level-environment, men ovenfor har vi ladet være åbent, hvordan denne værdi bestemmes.

Repræsentation af metoder (2).

```
(defun ensure-method (gf &rest all-keys)
  (let ((new-method
         (apply
           #'make-instance 'standard-method
           all-keys)))
    (add-method gf new-method)
    new-method))


(defun add-method (gf method)
  (let ((old-method
         (find-method gf (method-qualifiers method)
                     (method-specializers method) nil)))
    (when old-method (remove-method gf old-method)))
    (setf (method-generic-function method) gf)
    (push method (generic-function-methods gf))
    (dolist (specializer (method-specializers method))
      (pushnew method (class-direct-methods specializer)))
    (finalize-generic-function gf)
    method)
```


Noter

Ensure-metod skaber instansen af af standard-method, hvorefter add-method kaldes.

add-method fjerner først en evt. eksisterende metode fra den generiske funktion.

Dernæst etableres reference til den generiske funktion fra metoden.

Dernæst adderes metoden til den generiske funktion (jf. slotten method i standard-generic-function).

Dernæst tilføjes metoden til alle klasser, hvorpå metoden specialiserer (jf. slotten direct-metods i standard-class).

Endelig kaldes finalize-generic-function, som forestå forskellig bogholderi, som vi ikke er optaget af her og nu.

(**push** el place) betyder i Lisp-jargon at conse et element el på forenden af en liste, referet af place. Det er ækvivalet med

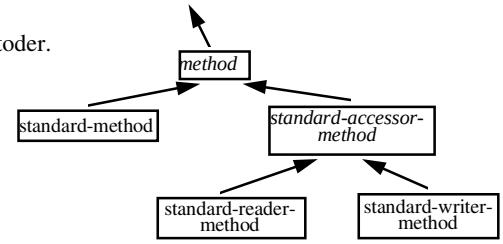
(setf place (cons el place))

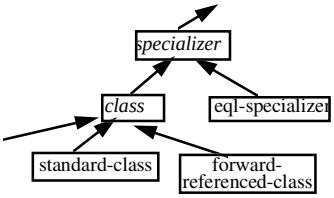
Pushnew conser på samme måde, men kun hvis el ikke allerede er i listen.

(**dolist** (el lst) krop) er et iterativt primitiv, der succesivt binder et element el i en liste lst, og udfører kroppen på denne. Mapning ville klart være at foretrække!

Hvad mere er metaobjekter?

- Reader-metoder, writer-metoder.
- Metodekombinationer.
- Slot-definitioner.
- Direkte slotdefinitioner.
- Effektive slotdefinitioner.
- Eql-specializers
- Forud refereret klasse.





Noter

Ud over at opremse de øvrige beskrivelser og elementer af CLOS, hvortil der findes metaklasser, og dermed metaobjekter (instanser af metaklasserne), giver denne figur også nogle dele af klassehierarkiet af metaklasserne, som ikke blev vist tidligere (hvor vi jo udelukkende koncentrerede os om metoder, klasser og generiske funktioner).

Kald af en generisk funktion (1).

(f a b) → (apply-generic-function #'f (list a b))

funktionen (generisk funktions metaobjekt), der er lagret under symbolet f.

```
(defun apply-generic-function (gf args)
  (let ((applicable-methods
         (compute-applicable-methods-using-classes
            gf (mapcar #'class-of  (required-portion gf args)))))
    (if (null applicable-methods)
        (error "There are no matching methods in the generic function")
        (apply-methods gf args applicable-methods))))

(defun compute-applicable-methods-using-classes (gf required-classes)
  (sort
      (copy-list
       ; don't consider those methods that are not applicable:
        (remove-if-not #'(lambda (method)
                           (every #'subclassp
                                  required-classes
                                  (method-specializers method)))
                       (generic-function-methods gf)))
    #'(lambda (m1 m2)
        (method-more-specific m1 m2 required-classes))))
```


Noter

apply-generic-function er "top niveau" funktion, der realiserer kald af en generisk funktion.

compute-applicable-methods-using-classes returnerer de anvendelige metoder *i sorteret rækkefølge*. Bemærk, at man kan bruge standard funktionen sort fra Common Lisp til denne sortering, såfremt man blot sender en funktion med som parameter, der angiver den indbyrdes ordning mellem to metoder (lambda formen, der sendes med som 2. parameter til sort).

(every #'subclassp required-classes (method-specializers method)) returnerer hvorvidt alle argumenternes klasser er subklasser af metode-specialiseringerne.

På næste side vil vi studere apply-methods.

Kald af en generisk funktion (2).

```
(defun apply-methods (gf args methods)
  (let ((primaries (remove-if-not #'primary-method-p methods))
        (befores (remove-if-not #'before-method-p methods))
        (reverse-afters (reverse (remove-if-not #'after-method-p methods))))
    (when (null primaries)
      (error "No primary methods")
    (dolist (before befores)
            (apply-method before args ()))
    (multiple-value-prog1
       (apply-method (car primaries) args (cdr primaries))
       (dolist (after reverse-afters))
         (apply-method after args ()))))))


(defun apply-method (method args next-methods)
  (let ((call-next-method  lambda som effekturerer et (call-next-method ) )
        (next-method-p   lambda som fortæller hvorvidt der er en 'next method' ))
    (eval (method-body method)
          (add-function-bindings
             '(call-next-method  next-method-p) (list call-next-method next-method-p)
              (add-varible-bindings
                 (method-lambda-list method)
                 args
                 (method-environment method))))))
```


Noter

Funktionen apply-methods (bemærk flertal) realiserer standard metodekombination, dog uden around methoder. Det skulle være lige ud ad landevejen at forstå, hvad der foregår her, når man har forstået standard metodekombination.

Funktionen apply-method (ental) anvender én enkelt metode. Det vi viser her er en noget idealiseret implementation, hvor vi antager vi har primitiver (add-function-binding, add-variable-binding), som kan producere et environment, hvori vi kan evaluere metodens krop.

Bemærk, at eval er Lisp's basale kald af sin egen fortolker.

‘Bootstrapping problemer.’

- Hvordan etableres metaobjekterne svarende til standard-class, standard-method, standard-generic-function mv.?
  - Klasser skabes ved brug af generiske funktioner (make-instance mv.).
  - Generiske funktioner kræver klasser, for at kunne lave ‘metode opslag’.
- Mulige løsninger:
  - instantiering af standard-class udføres ikke ved brug af make-instance, men via en speciel og mere primitiv teknik.
  - En del generiske funktioner, hvor det er forudseeligt på hvilke klasser de kaldes, realiseres som normale Common Lisp funktioner.

```
(defun ensure-class (name &rest all-keys)
 (if (find-class name nil)
     (error "...")
   (let ((class (apply (if (eq metaclass 'standard-class)
                           #'make-instance-standard-class
                           #'make-instance)
                       metaclass :name name all-keys)))
     (setf (find-class name) class)
     class)))
```


Noter

Specielt interesserede i disse problemer kan konsultere appendiks C i bogen *The art of the Metaobject Protocol* for detaljer.