

## Common Lisp

- Populärstes Lisp
- Großer Standard
- Mehrere kommerzielle Implementierungen
  - Große Bibliotheken
  - Hohe Geschwindigkeit (nahe C)
  - Gute IDEs

## CLOS

- Common Lisp Object System
- Sprich "kloss" oder "sieloss"
- OO-Erweiterung für Common Lisp
  - Klassenbasiert
  - Mehrfachvererbung
  - Generische Methoden
  - Meta Object Protocol

## Lisp

- Außergewöhnliche Syntax: Präfixnotation und volle Klammerung
- Funktionen sind Werte
- Hier keine Einführung sondern "learning by example"

Beispiel: Fakultätsfunktion

```
(defun fac (n)
  (if (= n 0)
      1
      (* n (fac (- n 1)))))

(fac 10) ;-> 3628800
```

## Definition von Klassen: defclass

Eine Klasse besteht aus mehreren Feldern, genannt *slots*

Eine Klasse kann mehrere Superklassen haben

```
(defclass cname ( superc ... ) slotspec ... )
```

## Slot-Spezifikation

Syntax: `slotspec = ( name option ... )`

Einige Optionen:

- `:initarg name` Keyword für Initialisierung
- `:reader name` Lesefunktion
- `:accessor name` Zugriffsfunktion für Lesen und Schreiben
- `:allocation :instance` oder `:class` Bestimmt, ob Instanz- oder Klassenslot

## Beispiel

```
(defclass position () ())

(defclass point (position)
  ((x :initarg :x :accessor point-x)
   (y :initarg :y :accessor point-y)))
```

## Instanzen erzeugen

Funktion `make-instance`

Argumente:

- Name der Klasse als Symbol (d.h. nach Hochkomma)
  - Werte für die Slots, jeweils hinter Keyword aus `:initarg`
- ```
(defvar origin (make-instance 'point :x 0 :y 0))
(defvar one (make-instance 'point :x 1 :y 1))
```

## Zugriff auf Slots

Stets über Funktionsaufrufe (keine `.-`-Syntax)

- Über Selektoren erzeugt von `:reader` `:accessor`
- Über das Primitivum `slot-value`

Setzen über `(setf (slot-accessor obj) expr)`

```
(setf (point-x origin) 2)
```

Also kein Zugriffsschutz

## Mehr zu defclass

Weitere Slot-Optionen:

- `:type t`  $\Rightarrow$  Typ des Arguments
- `:initform expr`  $\Rightarrow$  Default-Ausdruck für Wert
- `:writer name`  $\Rightarrow$  Mutatorfunktion
- `:documentation string`

`defclass` kann auch Klassenoptionen enthalten:

- `:metaclass class`  $\Rightarrow$  Angabe der Meta-Klasse
- `:documentation str`
- `:default-initargs expr...`  $\Rightarrow$  Default-Parameter für `make-instance`

## Verwendung für Slot-Optionen

- Kombiniere Slot-Optionen gemäß dieser Liste für `:initarg` `:reader` `:accessor`
- Nimm speziellsten Wert für `:initform`

## Mehrfachvererbung

CLOS bietet Mehrfachvererbung:

```
(defclass c (s1 .. sn) ...)
```

Immer problematisch: Wie werden Namenskonflikte aufgelöst?

Ansatz in CLOS: Erzeuge *class precedence list* aus Superklassendefinition

Idee: Speziellere Klassen stehen weiter vorne

- Aus Klassendefinition ergeben sich lokale Präzedenzen:  $c < s1$ ,  $s1 < s2$ , ...
- Nimm lokale Präzedenzen der Superklassen (und ihrer Superklassen) hinzu
- Bilde transitive Hülle
- Signalisiere Fehler, wenn Ergebnis keine partielle Ordnung ist
- Sortiere topologisch:  $c1$  muss vor  $c2$  vorkommen, wenn  $c1 < c2$

## Generische Funktionen

Beob: Klassen definieren nur Slots, keine Methoden

Grund: CLOS assoziiert Methoden nicht mit Klassen

Stattdessen: *Generische Funktionen*

- Funktion, die auf verschiedensten Argumenttypen anwendbar sind
- Implementierung einer gen. Funktion für spezielle Typen heißt dann *Generische Methode*

Alternative zu Message-Passing

Ebenfalls Dynamic-Dispatch

## Generische Funktionen definieren

Zumeist implizit durch Angabe einer generischen Methode

Syntax für generischen Methoden:

```
(defmethod name ( paraspec ... ) body )
```

*paraspec* ist dabei ein *Parameter-Specializer*

Syntax für *paraspec*:

- ( *name type* ): Name und Typ
- ( *name (eq1 obj)* ): Name und Wert
- ( *name t* ): Name und true = passt auf alles

## Beispiel

```
(defmethod dist ((a number) (b number)) (- a b))

(defmethod dist ((a point) (b point))
  (let ((xdist (dist (point-x a) (point-x b)))
        (ydist (dist (point-y a) (point-y b))))
    (sqrt (+ (* xdist xdist) (* ydist ydist))))

(defclass line (position)
  ((start :accessor line-start :type point)
   (end :accessor line-end :type point)))

(defmethod dist ((p point) (l line)) ...)

(defmethod dist ((l line) (p point)) (dist p l))

(dist origin one) ; -> 1.4142135
```

## Generische Funktionen aufrufen

Syntax wie bei normalen Prozeduraufrufen: ( *fexpr arg ...* )

Aufgerufen wird die Methode, deren Parameter-Specializer in der class precedence list den Argumenten am nächsten liegt

Bei mehreren Parametern: Lexikographische Ordnung

Aufgerufene Methode hat die Möglichkeit, mit `call-next-method` die nächst-allgemeinere Methode aufzurufen

## Vorteile von generischen Funktionen

- Dispatch nach mehreren Argumenten gleichzeitig (*multiple dispatch*)
- Generische Funktionen sind Werte erster Klasse
- Aufruf unterscheidet sich syntaktisch nicht von normalem Funktionsaufruf, da kein spezieller Receiver nötig ist
- Kein this/self nötig
- Verhalten kann unabhängig vom Klassengraphen spezifiziert werden

## Beispiel: Zahlen

```
(defclass rat ()
  ((num :accessor rational-num :initarg :num)
   (denom :accessor rational-denom :initarg :denom)))

(defmethod mult ((x integer) (y integer))
  (* x y))

(defmethod mult ((x rat) (y rat))
  (let ((num (* (rational-num x)
                (rational-num y)))
        (denom (* (rational-denom x)
                  (rational-denom y))))
    (make-instance 'rat :num num :denom denom)))
```

## Fortsetzung

```
(defmethod mult ((x rat) (y integer))
  (let ((num (* (rational-num x) y))
        (denom (rational-denom x)))
    (make-instance 'rat :num num :denom denom)))

(defmethod mult ((x integer) (y rat)) (mult y x))

(defmethod mult ((x t) (y (eql 1))) x)

(defmethod mult ((y (eql 1)) (x t)) x)
(mult 2 (make-instance 'rat :num 1 :denom 2))
; -> rat mit Wert 1
```

## Methodenkombination

Bis jetzt (Java, C++, Smalltalk, Objective-C, JavaScript) haben speziellere Methoden immer allgemeinere überschrieben

In CLOS gibt es zusätzlich *Hilfsmethoden* die *primären Methoden* erweitern

- Hilfsmethoden haben eine der Optionen `:before` `:after` `:around`
- `:around` kann mit `call-next-method` die primäre Methode aufrufen
- Alle passenden Hilfsmethoden werden in der Reihenfolge der class precedence list aufgerufen

Aufruf der generischen Funktion kombiniert dann die Methoden entsprechend

## Beispiel :before und :after

```
(defmethod dist :before ((p position) (p2 position))
  (print "Before dist for position"))

(defmethod dist :after ((p position) (p2 position))
  (print "After dist for position"))

(defmethod dist :before ((p point) (p2 point))
  (print "Before dist for point"))

(defmethod dist :after ((p point) (p2 point))
  (print "After dist for point"))

(dist origin one)
"Before dist for point"
"Before dist for position"
"After dist for position"
"After dist for point"
1.4142135
```

## Beispiel :around

```
(defmethod dist :around ((p position) (p2 position))
  (print "Begin around dist")
  (let ((result (call-next-method)))
    (print "End around dist")
    result))

(dist origin one)
"Begin around dist."
"Before dist for point"
"Before dist for position"
"After dist for position"
"After dist for pointn"
"End around dist"
1.4142135
```

## CLOS bisher

Klassenbasierte OO-Erweiterung für Common Lisp

Klassen haben nur Slots

Mehrfachvererbung und class precedence list

Generische Methoden mit multiple Dispatch

Methodenkombination erweitert Methodenaufruf um **:before**,  
**:after**, **:around**

## Anwendungen von Methodenkombination

- Logging von Funktionsaufrufen
- Caching von Funktionswerten

Allgemein: Sicherstellen/Hinzufügen von Vor- und Nachbedingungen

## Diskussion Methodenkombination

Vorteile:

- Ermöglicht verteilte Funktionsdefinitionen
- Deklarativ (im Gegensatz zu super-Aufrufen)

Nachteile:

- Mechanismus ist schwer zu durchschauen

## Das Meta-Object Protocol (MOP)

Das MOP ist die Spezifikation und Implementierung des CLOS

MOP ist selbst wieder objekt-orientiert (  $\Rightarrow$  Meta-Object)

Statischer Teil: Klassenhierarchie, deren Instanzen die CLOS-Klassen und -Methoden sind

Dynamischer Teil: Meta-Methoden beschreiben das Verhalten auf der Meta-Ebene

Besonderheit: Benutzer kann MOP durch eigene Klassen/Methoden anpassen

## Ziele von MOP

- Interna von CLOS verfügbar machen
- Programme sollen CLOS erweitern können

## Anwendungen

- Anpassung des Objekt-Systems an spezielle Anwendungen
- Hinzufügen von neuen Features
- Entfernen von Features
- Anderes Typsystem
- Persistenz
- Experimentieren mit OO-Systemen

## Statischer Teil von MOP

Repräsentation der Daten mittels Objekten

Meta-level Klassen für

- Klassen: **class**
- Slots: **slot-definition**
- Methoden: **method**
- Generische Funktionen: **generic-function**
- Methodenkombination: **method-combination**

## class

Instanzen sind die Klassen-Objekte

Wichtigste Meta-Klasse

Felder für alles, was `defclass` beschreibt (außer Slots):

- Direkte Superklassen
- Direkte Subklassen
- Präzedenzliste
- Initargs

Normale CLOS-Klassen sind Instanzen der Subklasse `standard-class`

## slot-definition

Instanzen enthalten die Informationen über die Slots wie bei `defclass` angegeben

- Name
- Type
- Initform
- ...

Instanzen von `standard-slot-definition` repräsentieren CLOS-Slots

## method

Methoden sind Instanzen von `method`

Gespeicherte Informationen:

- Methodenoptionen (:before, ...)
- Informationen über die Parameter

`standard-method` repräsentiert normale CLOS-Methoden

`standard-reader-method` und `standard-writer-method` repräsentieren die Methoden, die von `:reader` und `:accessor` erzeugt wurden

## Dynamischer Teil von MOP

Verhalten zur Laufzeit

Festgehalten in sog. Protokollen:

- Initialisierung von Klassen, Instanzen, Funktionen und Methoden
- Zugriff auf Slots
- Verwaltung von Abhängigkeiten
- Aufruf generischer Funktionen

## Bsp: Initialisierung von Klassen

**ensure-class** Erzeugt Klassen-Objekt

- **make-instance** erzeugt Objekt aus :metaclass-Option
- **validate-superclass** überprüft Superklassen
- **direct-slot-definition-class** Finde Klassen, die die Slots repräsentieren
- **make-instance** erzeugt Slot-Objekte
- **add-direct-subclass** fügt Klasse der Superklasse hinzu
- ...

## Bsp: Slot-Zugriff

**slot-value-using-class** Liefert Wert zu gegebenem Slot-Objekt

- **slot-exists-p** Beschreibt, ob Slot vorhanden ist
- **slot-missing** Wird aufgerufen, wenn Slot nicht vorhanden ist
- ...

analog für schreibenden Zugriff

## Bsp: Aufruf generischer Funktionen

**compute-discriminating-function** Berechnet CL-Prozedur für generische Funktion

- **compute-applicable-methods** Bestimmt alle möglichen Methoden
- **compute-effective-method** Kombiniert Methoden zu aufzurufender Methode

Die Klasse `standard-generic-function` spezifiziert Methodenaufruf für CLOS

Message-Passing wäre damit realisierbar wenn wirklich nötig

## Anwendung: Abstrakte Klassen

- Erzeugen von Instanzen soll nicht möglich sein
  - Ableiten soll möglich bleiben
- ⇒ Erzeugen neue Metaklasse **abstract-class**

## Metaklasse für abstrakte Klassen

```
(defclass abstract-class (standard-class)
  ()
  (:documentation "The class of abstract classes"))

(defmethod validate-superclass
  ((class abstract-class)
   (superclass standard-class))
  t)

(defmethod validate-superclass
  ((class standard-class)
   (superclass abstract-class))
  t)
```

## Erzeugen der Instanzen abfangen

`make-instance` erzeugt Instanz

`make-instance` ist generische Funktion

Erstes Argument ist Klassenobjekt

⇒ Generische Methode, spezialisiert auf `abstract-class` angeben

```
(defmethod make-instance ((c abstract-class)
                          &rest other-args)
  (error "Cannot make instance of abstract class"))
```

## Definieren von abstrakten Klassen

Beim Definieren immer `:metaclass`-Option für `defclass`

Kleines Schmankerl: Darüber abstrahieren mittels Makro

```
(defmacro define-abstract-class
  (class supers slots &rest options)
  `(defclass ,class ,supers ,slots
     ,@options
     (:metaclass abstract-class)))
```

## Einfachvererbung

Keine Mehrfachvererbung sondern nur Einfachvererbung

- Beim Erzeugen der Klassenobjekte Anzahl der Superklassen überprüfen
- Außerdem bei nachträglicher Veränderung der Klasse Anzahl der Superklassen überprüfen
- Nachprüfen, dass die Superklassen keine Mehrfachvererbung verwenden

## Metaklasse

Meta-Klasse für Klassen mit Einfachvererbung

```
(defclass single-inh-class (standard-class)
  ()))
```

## Superklassen überprüfen

Vereinfachung: Gültige Superklasse ist entweder oberste Klasse (`standard-object`) oder `single-inh-class`

```
(defmethod validate-superclass
  ((class single-inh-class)
   (superclass standard-class))
  nil) ;; superclass könnte MFV verwendet haben

(defmethod validate-superclass
  ((class single-inh-class)
   (superclass (eql (find-class 'standard-object))))
; t) ;; standard-object hat keine MFV
```

## Fortsetzung

```
(defmethod validate-superclass
  ((class single-inh-class)
   (superclass single-inh-class))
  t)

(defmethod validate-superclass
  ((class standard-class)
   (superclass single-inh-class))
  t) ;; erben darf jeder
```

## Erzeugen von Klassen überprüfen

Spezialisierung auf Instanzerzeugung von `single-inh-class`

```
(defmethod make-instance :around
  ((sih (eql (find-class 'single-inh-class)))
   &rest initargs
   &key direct-superclasses)
  (if (or (null direct-superclasses)
          (null (cdr direct-superclasses)))
      (call-next-method)
      (error "More than one superclass")))
```

## Nachträgliche Veränderung der Klasse

Eigentlich der Eintrittspunkt, aber bei Erstdefinition ist class=nil

Bei nachträglicher Veränderung ist die Klasse aber schon da:

```
(defmethod ensure-class-using-class :around
  ((class single-inh-class)
   (name t)
   &rest initargs
   &key metaclass direct-superclasses
   &allow-other-keys)
  (if (or (null (direct-superclasses))
          (null (cdr direct-superclasses)))
      (call-next-method)
      (error "more than one superclass"))))
```

## Nicht in CLOS

Keine Kapselung oder Zugriffsschutz: Aufgabe des Modulsystems

Einschränkung bei der Vererbung

## Außerdem noch in CLOS

- Argumentpräzedenzen für generische Methoden
- Optionen für generische Funktionen
- Unterstützung für Neudefinition von Klassen class-changed und change-class

## Zusammenfassung zu CLOS

- Klassenbasierte Sprache
- Als Erweiterung konzipiert
- Multiple-Dispatch als Alternative zu Message-Passing
- MOP als Programmierschnittstelle zum Objektsystem