Praktische Algorithmen der Bioinformatik und Computerlinguistik mit Lisp

Teil 8: Objektorientierte Programmierung mit CLOS

15. 6. 2021

LISP-Intensivkurs

Übersicht:

Objektorientierte Programmierung mit CLOS

- Einführung und Begriffsbestimmung
- Klassen
- Instanzen
- Methoden
- Methodenkombinationen

Grundzüge Objektorientierter Programmierung

Versuch einer Definition von P. Wegner 1987:

Objektorientierung
=
Objekte + Klassen + Vererbung

Objekte haben einen *Zustand*, dargestellt durch Variablen, und ein *Verhalten*, dargestellt durch **Methoden**. Hierbei ist die Idee des *information hidings* wichtig, bei dem die Manipulation des Zustands *nur* über ausgewählte Methoden des Objektes selbst realisiert wird.

Grundzüge Objektorientierter Programmierung

Objekte werden aufgrund von Ähnlichkeiten bzgl. Struktur und Verhalten gruppiert zu **Klassen**. Ein Objekt einer Klasse heißt dann *Instanz* dieser Klasse. Definition einer Klasse umfaßt:

- Variablen (Instanz/Klasse, Anfangswerte,...)
- Methoden
- Vererbungsrelation

Grundzüge Objektorientierter Programmierung

Mittels **Vererbung** können Varianten existierender Klassen definiert werden. Dies führt zu einer *Klassenhierarchie*. Die neue Klasse *erbt* Verhalten (z.B. Methoden) der *Elternklasse(n)*; somit muß nur der Unterschied zu dieser Elternklasse(n) programmiert werden. Bei Vererbung von mehreren Elternklassen spricht man auch von *multipler Vererbung*.

Auswahl objektorientierter Sprachen

- C++, Java, Simula, Smalltalk, ...
- CommonLOOPS, Flavors, CLOS,...

Grundzüge Objektorientierter Programmierung

Einfluß objekt-orientierter Vorgehensweise auf die Code-Qualität bezüglich:

Korrektheit

Klar definierte “Module” mit Interface; dies erleichtert es, diese “Module” als korrekt nachzuweisen, sowie Interaktionen zwischen Modulen zu analysieren.

Vorsicht: CLOS unterstützt diese Form des *Information-hiding* nicht!

Robustheit

Generische Funktionen ermöglichen es zum Beispiel, daß eine Funktion zur Laufzeit Argumente von einem Typ akzeptiert, der beim Schreiben der Funktion selbst noch gar nicht definiert wurde. Beispiel: Default-Methoden.

Grundzüge Objektorientierter Programmierung

Einfluß objekt-orientierter Vorgehensweise auf die Code-Qualität bezüglich (Fortsetzung):

Erweiterbarkeit

Objekt-orientierte Systeme mit Vererbungsmechanismen erlauben es, neue, leicht abgewandelte Klassen zu definieren und ohne weiteren Aufwand in das existierende System mit einzubinden.

Wiederverwendbarkeit

Im objekt-orientierten Paradigma benutzt man entweder existierende Klassen einer Bibliothek, oder aber man spezialisiert existierende Klassen durch Vererbung.

Grundzüge Objektorientierter Programmierung

Einfluß objekt-orientierter Vorgehensweise auf die Code-Qualität bezüglich (Fortsetzung):

Kompatibilität

Eine objekt-orientierte Vorgehensweise unterstützt den Aufbau von Bibliotheken. Je mehr Programme Standardkomponenten benutzen desto mehr Kommunikation wird zwischen diesen Programmen möglich sein.

Programmieren in CLOS

- Definition von **Klassen**, die Daten nach Komponenten und Charakteristika organisieren
- Bilden von **Objekten** (oder:Instanzen) dieser Klassen; Struktur und Verhalten von Instanzen wird bestimmt durch ihre Klassen;
- Definition von **Methoden**, um klassenspezifische Operationen auszuführen
- Erweitere Charakteristika von Klassen durch **Vererbung**

Einführendes Beispiel

Klassen-Definition

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

Einführendes Beispiel

Generierung von Objekten

```
(setf r1 (make-instance 'rectangle :start-height 50
                                   :start-width 100))
```

Zugriff auf Slots

```
(rectangle-height r1)
(setf (rectangle-height r1) 75)
```

Einführendes Beispiel

Strukturvererbung

```
(defclass color-rectangle (rectangle)
    ((color :initform 'red
            :initarg :color
            :accessor color)
     (clearp :initform (y-or-n-p "Transparent?")
             :initarg :clearp
             :accessor clearp)
     (height :initform 10)))
```

Instanzen von `color-rectangle` haben vier Slots:

Neu	`color, clearp`
Vererbt	`height, width`

Einführendes Beispiel

Problem: Was passiert bei sich widersprechender neuer und vererbter Information?

Ansatz: Grundsätzlich überschreibt spezifischere Information die generellere oder vererbte.

Einführendes Beispiel

Beispiel für **multiple Vererbung** (engl. multiple inheritance).

```
(defclass color-mixin ()
   ((color :initform 'red
           :initarg :color
           :accessor color)))

(defclass color-rectangle-2 (color-mixin rectangle)
   ((clearp :initform nil
            :initarg :clearp
            :accessor clearp)
    (height :initform 10)))
```

Einführendes Beispiel

- Mixin-Klassen zur Beschreibung unabhängiger Fragmente von Strukturen oder Verhalten;
- Instanziierbare Klassen werden dann meistens mit Hilfe mehrerer Mixins und einer Hauptklasse (sog. *principal class*) gebildet.

Bemerkung: multiple Vererbung ist in vielen objekt-orientierten Programmiersprachen nicht möglich.

Generische Funktionen und Methoden

Beispiel:

```
(defgeneric paint (shape medium)
   (:documentation "Malt etwas auf etwas anderes"))
```

Eine **generische Funktion** spezifiziert die gemeinsame Schnittstelle von Methoden, insbesondere den generischen Funktionsnamen und die Argumentliste; bei Aufruf einer generischen Funktion legt ein Laufzeit-Dispatcher eine Kombination von Methoden fest, die ausgeführt werden.

Generische Funktionen und Methoden

```
(defmethod paint ((shape rectangle) ;; Methode-1
                  medium)
   (format t "~& Somehow paint rect. ~a on medium ~a"
      shape medium))

(defmethod paint ((shape color-rectangle) ;; Methode-2
                  medium)
   (format t "~& Somehow paint color rect. ~a on ~a"
      shape medium))
```

Generische Funktionen und Methoden

Aufruf der generischen Funktion paint:

```
(paint r1 *standard-display*)
```

resultiert in den Aufruf von `Methode-1`, da `r1` vom Typ `rectangle` ist (s.o.). Welche Methode aufgerufen wird, hängt also vom Typ des Argumentes ab.

Methoden-Kombination

Aufruf einer generischen Funktion (oder: Senden einer Message) resultiert möglicherweise in eine Sequenz von Methodenaufrufen.

```
(defmethod paint
  :before ((shape color-rectangle) medium)
  (format t "Prepare brush for painting color rect. ~a"
     shape))

(defmethod paint :after ((shape color-rectangle) medium)
   (format T "Clean brush"))
```

Sei nun etwa:

```
(setf cm (make-instance 'color-rectangle))
```

Methoden-Kombination

Beim Aufruf von `paint` mit einem Argument der Klasse `color-rectangle` (oder evtentuell einer Spezialisierung davon) wird zuerst die `:before` Methode, danach die Primärmethode und zuletzt die `:after` Methode der generischen Funktion paint aufgerufen.

```
> (trace paint)

> (paint cm)

ENTER ...
EXIT  ...
```

Das Resultat des Funktionsaufrufs der generischen Funktion berechnet sich aus dem Resultat der Primärmethode.

Klassen-Deklaration

Syntax des `defclass` Makros:

```
(defclass class-name (supercl-1 ... supercl-n)
   ((slot-name-1 slot-options-1)
    ...
    (slot-name-m slot-options-m))
   (:documentation "..."))
```

Bemerkung: jede Klasse definiert einen CL Datentyp

Beispiel:

```
(defclass automobile ()
   ((doors :initarg :doors :accessor doors)
    (color :initarg :color :accessor color)))

(defclass sedan (automobile)
   ((engine-type :initarg :engine-type)))
```

Klassen-Deklaration

Slot-Spezifikationen:

`:reader`
`:writer`
`:accessor`
	Zugriffsmethoden

`:allocation` `:instance`	lokaler Slot für jedes Objekt der Klasse allokiert (default)
`:allocation` `:class`	genau ein Slot für alle Objekte der Klasse gemeinsam
`:initform` `:initarg`	Konstruktoren
`:type`	spezifiziert einen Datentyp für einen Slot

Klassen-Deklaration

Class Precedence List (CPL)

Kontrolle der Vererbung von Slots, Slot Optionen und Präzedenz von Methoden

- vollständige Ordnung einer Klasse und ihrer Superklassen;
- Ordnung von spezifisch zu weniger spezifisch;
- Algorithmus zur Berechnung der CPL beachtet die Regeln

  Regel 1 Klassen kleiner als ihre Superklassen

  Regel 2 eine direkte Superklasse ist kleiner als alle anderen Superklassen zur rechten in der Superklassen-Deklaration in defclass;
- Diese beiden Regeln reichen noch nicht aus, um eine eindeutige Ordnung zu charakterisieren;
- Klassen-Hierarchie: gerichteter azyklischer Graph(DAG)

Klassen-Deklaration

Beispiel für CPL

```
(defclass C1 () ())
(defclass C2 (C1) ())
(defclass C3 (C1) ())
(defclass C4 (C3) ())
(defclass C5 (C3) ())
(defclass C6 (C4 C5) ())
```

Somit:

```
CPL(C6) = <C6,C4,C5,C3,C1,standard-object,t>
```

Dies läßt sich auch mit dem Inspektor feststellen:

```
(inspect (find-class 'C6))
```

Klassen-Deklaration

Beispiel:

```
(defclass A (B) ())
(defclass B (C) ())
(defclass C (A) ())
```

-> Zyklus (Sollte Fehler erzeugen)

Klassen-Deklaration

Vererbung von Slots und Slot-Optionen wird über die CPL gesteuert.

Vererbung von Slots:

- Klassen erben Slots von ihren Superklassen.
- Falls ein Slot mehrmals in den Superklassen deklariert wurde, so müssen die verschiedenen Slot-Optionen kombiniert werden.

Klassen-Deklaration

Kombination von defclass **Slot-Optionen:** Entweder die spezifischere (bzgl. CPL) Information überschreibt die allgemeinere, oder aber die Information wird “vereinigt”.

`:reader`	Vereinigung
`:writer`	Vereinigung
`:accessor`	Vereinigung
`:initform`	Überschreibung
`:allocation`	Überschreibung
`:initarg`	Vereinigung
`:type`	Durchschnitt: `(and type-1 ... type-n)`

Bemerkung: Kombination von :reader, :writer und :accessor folgt aus der normalen Methodenkombination.

Klassen-Deklaration

Beispiel:

```
(defclass C1 ()
  ((S1 :initform 2.3 :type number)
   (S2 :allocation :class)
   (S4 :accessor C1-S4)))

(defclass C2 (C1)
  ((S1 :initform 5 :type integer)
   (S2 :allocation :instance)
   (S3 :accessor C2-S3)))

(describe (find-class 'C1))  ==> ...
(describe (find-class 'C2))  ==> ...
```

Klassen-Deklaration

Bemerkung: Bei multipler Vererbung wird nach Definition der CPL immer von der am weitesten links stehenden Klasse überschrieben; z.B ist in

```
(defclass A (B C) ...)
```

Die Klasse `B` spezifischer als `C`.

Klassen-Deklaration

Beispiel für multiple Vererbung

```
(defclass motor-vehicle ()
   ((engine-type :initarg :engine-type
                 :accessor engine-type)
    (wheels :allocation :class
            :initform 4
            :accessor wheels)))

(defclass house ()
   ((rooms :initarg :rooms :accessor rooms)
    (stove :initform 'electric :initarg :stove)))

(defclass motor-home (motor-vehicle house) ())

(setf bago (make-instance 'motor-home
             :engine-type 'V8
             :rooms 3
             :stove 'propane))
```

Klassen-Deklaration

```
(describe bago)
#<Motor-Home #X1AB24D6> is an instance of the
 class MOTOR-HOME:
The following slots have allocation :INSTANCE:
ROOMS              3
STOVE              PROPANE
ENGINE-TYPE        V8
The following slots have allocation :CLASS:
WHEELS             4
```

Integration von CLOS in CL

- einige vordefinierte CL Typen haben eine korrespondierende Klasse mit demselben Namen wie der Typ
- Namen von Klassen sind CL Typ- Spezifikatoren

Einige CL-Funktionen funktionieren auch auf CLOS-Objekten

```
(typep object class) ==> t
```

falls Klasse von object Subklasse von class oder Klasse von `object` ist `class` selbst.

```
(subtypep class1 class2) ==> t
```

falls Klasse `class2` hat `class1` als Subklasse oder `class1` und `class2` identisch sind.

```
(type-of object)
```

liefert (implementierungsabhängig?) den Typ von `object`.

Integration von CLOS in CL

Class	Class Precedence List
bignum	(bignum integer rational number t)
clos:array	(clos:array t)
bit-vector	(bit-vector vector clos:array sequence t)
character	(character t)
complex	(complex number t)
cons	(cons list sequence t)
double-float	(double-float float number t)
fixnum	(fixnum integer rational number t)
float	(float number t)
hashtable	(hashtable t)
integer	(integer rational number t)

Integration von CLOS in CL

Class	Class Precedence List
list	(list sequence t)
null	(null symbol list sequence t)
number	(number t)
package	(package t)
readtable	(readtable t)
random-state	(random state t)
ratio	(ratio rational number t)
rational	(rational number t)
sequence	(sequence t)
single-float	(single-float float number t)
string	(string vector clos:array sequence t)
symbol	(symbol t)
t	(t)
vector	(vector clos:array sequence t)

Instanzen von Klassen

```
(make-instance class-name
               :arg-1 value-1
               ...
               :arg-n value-n)
```

Beispiel:

```
(setf her-sedan (make-instance 'sedan :doors 4
                                      :color 'red
                                      :engine-type 'V6))
```

Konstruktoren: Es ist oft guter Stil, daß man eigene Konstruktoren definiert.

```
(defun make-sedan (doors color engine-type)
   (make-instance 'sedan :doors doors
                         :color color
                         :engine-type engine-type))
```

Zugang zu Slots und Slot-Werten

vordefinierte Zugriffsmethoden durch `defclass`

```
(doors her-sedan)            ==> 4
(setf (doors her-sedan) 3) ==> 3
(doors her-sedan)            ==> 3
```

alternativ:

```
(slot-value her-sedan 'doors)            ==> 4
(setf (slot-value her-sedan 'doors) 3) ==> 3
(slot-value her-sedan 'doors)            ==> 3
```

Instanzen von Klassen

Weitere Methoden zur Manipulation von Slots

```
(slot-boundp instance slot-name)
```

falls `slot-name` nicht existiert, wird `slot-unbound` aufgerufen.

```
(slot-exists-p object slot-name)
```

```
(slot-makunbound instance slot-name)
```

falls `slot-name` nicht existiert, wird `slot-missing` aufgerufen.

Kontrolle von Instanzen

Anzeige von Objekten

```
(print-object object stream)
```

Man schreibt sich seine eigenen print-object Methoden, um die Ausgabe von Objekten zu steuern; zum Beispiel:

```
(defmethod print-object ((car sedan) stream)
   (format stream "#<~S ~A ~D>"
      (type-of car)
      (if (slot-boundp car 'doors)
         (doors car)
         "unknown")
      (color car)))
```

Kontrolle von Instanzen

Kontrolle über die **Instanziierung** von Objekten mit Hilfe von **`:after`** Methoden zu

```
(initialize-instance instance &rest initargs)
```

Denn `initialize-instance` wird aufgerufen von `make-instance`.

Generische Funktionen

Generische Funktionen in CLOS korrespondieren zu *Nachrichten* (engl. messages) in anderen objektorientierten Sprachen.
generische Funktionen sind Lisp-Funktionen

ALSO: funcall,apply auf generische Funktionen möglich
Unterschiede zu gewöhnlichen CL-Funktionen:

- Operation ist klassen-spezifisch
- benützt bei der Ausführung eine Menge von Methoden

Definition generischer Funktionen:

```
(defgeneric function-specifier (par-1 ... par-n)
  (:method method-1)
  ...
  (:method method-m)
  (:documentation "..."))
```

Generische Funktionen

Beispiel:

```
(defgeneric foo (x)
   (:method ((x string)) (princ "it's a string"))
   (:method :before ((x string)) (princ "Hey Dude, "))
   (:method ((x sequence)) (princ "it's a sequence"))
   (:method ((x integer)) (princ "it's an integer"))
   (:documentation "nonsense function"))

(foo 4)                    ==> "it's an integer"
                           Seiteneffekt: it's an integer
(foo "hallo")              ==> "it's a string"
                           Seiteneffekt:
                                  Hey Dude, it's a string
(foo '(a b c))             ==> "it's a sequence"
                           Seiteneffekt: it's a sequence
(documentation #'foo)      ==> "nonsense function"
```

Generische Funktionen

“Aufruf” einer generischen Funktion:

- anwendbare Methoden werden ausgewählt
- Sequenz der Methoden- Ausführung wird festgelegt
- Methoden werden ausgeführt
- Rückgabewert der generischen Funktion wird aus den Rückgabewerten der Methoden berechnet

Bemerkung: Oftmals werden die Methoden zu einer generischen Funktion nicht direkt mit in der Definition der generischen Funktion mitdefiniert. Für separate Definition von Methoden gibt es das `defmethod` Makro.

Methode

Methoden führen die klassen-spezifischen Operationen einer generischen Funktion aus. Ein Methoden-Objekt ist keine Lisp-Funktion!

```
(defmethod function-specifier {method-qualifier}*
     ((var-1 class-1)...var-n...
      (var-m (eql value)))
     ...lisp code...)
```

Bemerkung: Bei Methoden unterscheidet man zwischen

- Primärmethoden(engl. primary methods)und
- Hilfsmethoden(engl. auxiliary methods)
  - `:after`
  - `:before`

Diese Unterscheidung kommt bei der Kombination von Methoden, wie sie von generischen Funktionen vorgenommen wird, zum Tragen.

Methode

Bemerkung: `defmethod` deklariert implizit eine generische Funktion, falls eine solche noch nicht deklariert wurde:

```
(defmethod add-one ((x number)) (+ x 1))
```

Methode

Bemerkung: Lambda-Listen einer generischen Funktion und aller ihrer Methoden müssen *kompatibel* sein. Dabei heißen, im vereinfachten Falle, zwei Lambda-Listen kompatibel, falls sie die gleiche Anzahl von Parametern haben (wir ignorieren hier &rest, &optional und &key Parameter)

Anwendbarkeit von Methoden: Eine Methode ist anwendbar, falls jeder der benötigten Parameter erfüllt wird durch das korrespondierende Argument `(arg)` der generischen Funktion.

formaler Parameter	Test
`(var-name class-name)`	`(typep arg 'class-name)`
`var`	`(typep arg 't)`
`(var-name (eql form))`	`(eql arg 'form)`

Methode

Sortieren von Methoden: von spezifisch nach weniger spezifisch bzgl. CPL.

Ausführung von Methoden: Es wird (im vereinfachten Falle) die spezifischste Methode ausgeführt.

Methode

Beispiel:

```
(defclass C1 () ())
(defclass C2 (C1) ())
(defclass C3 (C2) ())
(setf c3 (make-instance 'C3))

(defmethod f ((x C1)) "Methode C1")
(f c3)   ==> "Methode C1"

(defmethod f ((x C2)) "Methode C2")
(f c3)   ==> "Methode C2"

(defmethod f ((x C3)) "Methode C3")
(f c3)   ==> "Methode C3"
```

Methoden-Kombination

Standard-Methoden-Kombination: `:before` Methoden, Primärmethoden und `:after` Methoden werden in der folgenden Reihenfolge aufgerufen:

- `:before` Methoden werden von spezifisch (bzgl. *CPL* zu weniger spezifisch aufgerufen.
- die bezüglich *CPL* spezifischste Primärmethode wird ausgeführt;
- `:after` Methoden werden von weniger spezifisch nach spezifischer aufgerufen.
- Wert eines Aufrufs einer generischen Funktion ist (in unserem vereinfachten Falle) der Wert der aufgerufenen Primärmethode.

Methoden-Kombination

Beispiel:

```
(defclass C1 () ())
(defclass C2 (C1) ())
(defclass C3 (C2) ())

(setf c3 (make-instance 'C3))
```

Methoden-Kombination

```
(defmethod f ((x C1)) (princ "Primaer C1 "))

(f c3)  ==>  "Primaer C1"
        Seiteneffekt: Primaer C1

(defmethod f ((x C2)) (princ "Primaer C2 "))

(f c3)  ==> "Primaer C2"
        Seiteneffekt: Primaer C2

(defmethod f ((x C3)) (princ "Primaer C3 "))

(f c3)  ==> "Primaer C3"
        Seiteneffekt: Primaer C3

(defmethod f :before ((x C1)) (princ "Before C1 "))
(defmethod f :after ((x C1)) (princ "After C1 "))
```

Methoden-Kombination

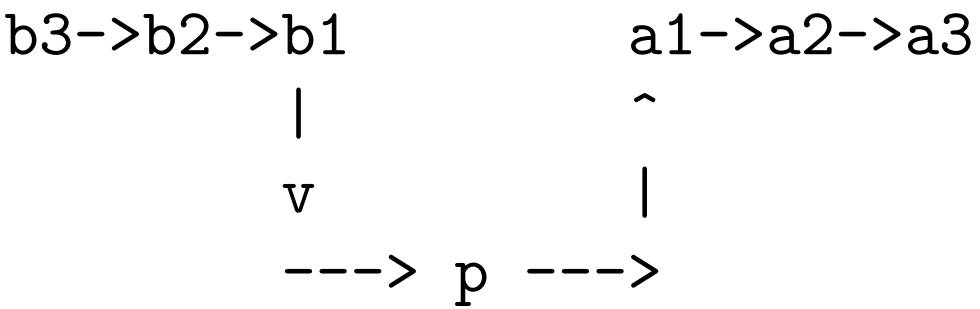
```
(f c3)  ==> "Primaer C3"
        Seiteneffekt: Before C1 Primaer C3 After C1

(defmethod f :before ((x C2)) (princ "Before C2 "))
(defmethod f :before ((x C3)) (princ "Before C3 "))
(defmethod f :after ((x C2)) (princ "After C2 "))
(defmethod f :after ((x C3)) (princ "After C3 "))

(f c3)  ==> "Primaer C3"
        Seiteneffekt: Before C3 Before C2 Before C1
                      Primaer C3
                      After C1 After C2 After C3
```

Methoden-Kombination

Damit ergibt sich also folgendes Bild:



Außerdem gibt es bei der Methodenkombination:

- `:around` Methoden
- Aufruf weiterer Primärmethoden mit call-next-method
- neben der Standardmethodenkombination (s.o.) weitere Methodenkombinationen, die durch den Methodenkombinationstyp einer generischen Funktion angewandt werden können;
- Es lassen sich auch eigene Methodenkombinationstypen vom Programmierer definieren (nicht in Lucid 4.0.1 aber in Genera 8.0)

Zugriffs-orientierte Programmierung

- es werden Methoden an Slots angehängt, die bei Schreib- oder Lesezugriffen aktiviert werden
- Neuberechnung eines Slots kann verzögert werden, bis das Ergebnis gebraucht wird
- eine neue Belegung von Slots kann Methoden aktivieren

Beispiel: jedesmal wenn sich die Höhe oder Länge eines Vierecks ändert, soll die Fläche berechnet werden. Die Frage ist dann nur, ob die Fläche *sofort* beim Schreiben eines neuen Wertes berechnet wird, oder aber erst *verzögert* bei einem Zugriff auf die Fläche des Vierecks. Ersteres erreicht man durch **Spezialisierung** von Schreibmethoden, und das zweite durch Spezialisierung von Lesemethoden.)

Zugriffs-orientierte Programmierung

```
(defmethod rectangle-area
  :before ((shape rectangle))
  (setf (rectangle-area shape)
      (* (rectangle-height shape)
         (rectangle-width shape))))

(rectangle-area r1)                ==> 24

(setf (rectangle-width r1) 7)

(slot-value r1 'area)              ==> 24

(rectangle-area r1)                ==> 28
```

Zugriffs-orientierte Programmierung

Spezialisierung von Lesemethoden:

```
(defmethod rectangle-area
  :before ((shape rectangle))
  (setf (rectangle-area shape)
      (* (rectangle-height shape)
         (rectangle-width shape))))

(rectangle-area r1)                ==> 24

(setf (rectangle-width r1) 7)

(slot-value r1 'area)              ==> 24

(rectangle-area r1)                ==> 28
```

Zugriffs-orientierte Programmierung

Spezialisierung von Schreibmethoden:

```
(defmethod (setf rectangle-height)
  :after (y (shape rectangle))
  (setf (rectangle-area shape)
      (* (rectangle-height shape)
         (rectangle-width shape))))

(defmethod (setf rectangle-width)
  :after (y (shape rectangle))
  (setf (rectangle-area shape)
      (* (rectangle-height shape)
         (rectangle-width shape))))

(setf  (rectangle-height r1) 3)
(slot-value r1 'area)    ==> 21

(setf  (rectangle-width r1) 9)
(slot-value r1 'area)    ==> 27
```

Individuelle Methoden

mind. ein Parameter-Spezialisierer ist von der Form
`(var-name (eql object))`

Beispiel: Abfangen von Division durch 0 mittels einer individuellen Methode.

```
(defmethod divide
  ((dividend number) (divisor number))
  (/ dividend divisor))

(defmethod divide
  ((dividend number) (zero (eql 0)))
   :infinity)

(divide 5 3)   ==> 5/3

(divide 3 0)   ==> :infinity
```

Individuelle Methoden

Multimethoden sind dadurch gekennzeichnet, daß sie mehr als einen Parameter-Spezialisierer besitzen.

Beispiel: Definition von conc, das eine Art append auf Listen *und* Vektoren realisiert.

```
(defmethod conc ((x null) y) y)

(defmethod conc (x (y null)) x)

(defmethod conc ((x list) (y list))
   (cons (first x) (conc (rest x) y)))

(defmethod conc (( x vector) (y vector))
  (let ((vect (make-array (+ (length x) (length y)))))
    (replace vect x)
    (replace vect y :start1 (length x))))
```

Multimethoden

Beachte, daß im Falle eines nil Argumentes, zwei Methoden anwendbar sind; es wird aber die Methode für `null` aufgerufen, da diese Klasse spezifischer ist als list.

```
(conc nil '(a b c))          ==>   (A B C)
(conc '(a b c) nil)          ==>   (A B C)
(conc '(a b c) '(d e f)      ==>   (A B C D E F)
(conc '#(a b c) '#(d e f))   ==>   #(A B C D E F)
```

Multimethoden sind *anwendbar*, falls jeder Parameter-Spezialisierer vom zugehörigen Parameter der generischen Funktion erfüllt wird.

Multimethoden werden geordnet, indem Parameter von links nach rechts betrachtet werden: lexikographische Ordnung.

Multimethoden

Bemerkung: Es ist in vielen Fällen unintuitiv, daß die Reihenfolge der Parameter die Ordnung auf Methoden bestimmt.

Bemerkung: Der Stil von Methoden-Definitionen ist ähnlich dem von “Funktions”-Definitionen in funktionalen Sprachen: irgendeine *funktionale Programmiersprache* (Haskell, ML, Miranda)

```
datatype 'a list = nil | cons of 'a * ('a list);

len nil          = 0;
len cons(a,l)    = 1 + len l;
```

Abstrakte Datentypen in CLOS

In CLOS sind Datentypen `list` zusammen mit Untertypen `null` und cons vordefiniert. Mit dieser Typhierarchie lassen sich jetzt Funktionen über den Aufbau von Listen in einer ähnlichen Weise wie bei funktionalen Programmiersprachen schreiben (es gibt aber bisher noch kein *Pattern-Matching*).

```
(defmethod len ((x null)) 0)

(defmethod len ((ll cons))
   (let ((a (first ll))
         (l (rest ll)))
     (1+ (len l))))
```

Verschiedenes

Kontrolle über Fehlersituationen

```
(no-applicable-method generic-function
                      &rest function-args)

(no-next-method generic-function calling-method
                      &rest args)

(slot-missing class instance slot-name operation
                      &optional new-value)

(slot-unbound class instance slot-name)
```

Verschiedenes

- Dynamische Modifikation generischer Funktionen, Methoden, Klassen, Instanzen
- Meta-ObjektProtokoll;
  - noch nicht vollständig standardisiert
  - noch keine vollständigen und effizienten Implementierungen

z.B. Auflisten aller Slot-Namen einer Klasse:

```
(defun class-slot-name (class-name)
  (mapcar #'clos:slot-definition-name
    (clos:class-slots (find-class class-name))))
```

Techniken: Methoden vs. Kontrollstrukturen

Rückblende: Anstelle eines `cond`-Konstrukts ist die Fallunterscheidung durch `case` oft bequemer:

```
> (case (second '(1 2 3))
        (1 "eins")
        (3 "drei")
        ((2 4) "gerade Zahl")
        (T "zu gross"))
"gerade Zahl"
```

Techniken: Methoden vs. Kontrollstrukturen

In analoger Art kann mit `typecase` eine Fallunterscheidung über Typen durchgeführt werden:

```
> (let* ((val '(1 2))
         (typename
           (typecase val
                     (number "Zahl")
                     (list "Liste")
                     (string "Zeichenkette")
                     (T "sonstwas"))))
          (format T "Ausgabe von ~A ~A" typename val))

Ausgabe von Liste (1 2)
NIL
```

Aufgabe: Erklären Sie, warum `(typecase val ...)` nicht leicht durch `(case (type-of val) ...)` simuliert werden kann. Schreiben Sie dazu obiges Beispiel entsprechend um und vergleichen Sie!

Techniken: Methoden vs. Kontrollstrukturen

Wenn eine Fallunterscheidung nach Typen vorgenommen werden soll, kann (!) ein `typecase`-Konstrukt günstiger sein als eine Menge von Methoden:

```
(defclass c0 () ((c0-slot :accessor c0-slot)))
(defclass c1 (c0) ((c1-slot :accessor c1-slot)))
(defclass c2 (c0) ((c2-slot :accessor c2-slot)))

(defun complicated-output (obj)
  (format T "Hier kommt eine lange Ausgabe,~%")
  (format T "die sich nur durch ein Detail: ~A"
          (typecase obj
                    (c1 (c1-slot obj))
                    (c2 (c2-slot obj))
                    (c0 (c0-slot obj))))
  (format T " unterscheidet"))
```

Techniken: Methoden vs. Kontrollstrukturen

```
> (complicated-output
      (let ((inst (make-instance 'c0)))
           (setf (c0-slot inst) 2) inst))
Hier kommt eine lange Ausgabe,
die sich nur durch ein Detail: 2 unterscheidet
```

Dies kann übersichtlicher und schneller sein als:

```
(defmethod complicated-output ((obj c0))
  (format T "Hier kommt eine lange Ausgabe,~%")
  (format T "die sich nur durch ein Detail: ~A"
          (c0-slot obj))
  (format T " unterscheidet"))
(defmethod complicated-output ((obj c1)) ...)
(defmethod complicated-output ((obj c2)) ...)
```

Zusammenfassung

Objektorientierte Programmierung ist im allgemeinen charakterisiert durch:

- *Klassen*, die eine Menge von Objekten beschreiben
- Vererbungsbeziehungen zwischen Klassen
- *Objekte*, die u.U. mehreren Klassen zugeordnet werden können
- *Methoden*, die auf Objekten arbeiten

CLOS bietet speziell die Möglichkeit von:

- *Multipler Vererbung:* Eine Klasse kann von mehreren anderen Klassen erben
- *Multimethoden:* Eine Methode kann durch mehrere Klassen spezialisiert werden
- Methoden können mit before- und after- Methoden versehen werden

Dadurch werden verschiedene Konfliktlösungsstrategien notwendig.