Ausarbeitung zum Thema Common Lisp/CLOS

Katrin Lang

Compilerbau-Seminar „Typsysteme“ bei Prof. Peter Pepper 10.02.2006
Fachgebiet Übersetzerbau und Programmiersprachen
Technische Universität Berlin

langk@cs.tu-berlin.de

Zusammenfassung:

Lisp ist eine dynamisch getypte Sprache, die funktionale, prozedurale und seit einiger Zeit auch objektorientierte Programmierparadigmen in sich vereint.
Gegenstand dieser Ausarbeitung ist das Typsystem des Common Lisp-Dialekts und insbesondere dessen Objektsystem mit einem kurzen Ausblick auf das Metaobjekt-Protokoll.

Inhalt:

1 Einleitung....................................2
2 Dynamische vs. Statische Typisierung....................................2
3 Das Common Lisp Object System.................. 3
  3.1 Integration....................................4
  3.2 Klassen....................................5
  3.3 Generische Funktionen....................................9
  3.4 Methoden....................................10
  3.5 Methoden-Kombination....................................12
4 Das Metaobjekt-Protokoll....................................14
5 Quellen....................................16

1 Einleitung

Lisp wurde ursprünglich Ende der 1950er Jahre am MIT entwickelt und war die erste Sprache, die funktionale Programmierparadigmen einführte. Sie hat dahingehend viele spätere funktionale Sprachen inspiriert, ist aber keine rein funktionale Sprache, da sie u.A. mit Seiteneffekten arbeitet.
Lisp ist eine typsichere, dynamisch getypte Sprache. Jedoch überlässt Lisp dem Programmierer Freiheiten in der Auswahl zwischen den beiden Extremen Effizienz und Sicherheit. Kapitel 2 wird darauf näher eingehen. Kapitel 3 stellt Philosophie und Historie des Common Lisp Object Systems(CLOS) und seine Integration in Common Lisp vor (Kap 3.1). Die weiteren Unterkapitel beschäftigen sich mit der Implementierung auf Grund von Klassen (Kap. 3.2) und generischen Funktionen (Kap. 3.3). Dieser Ansatz unterscheidet sich von Message-Passing dadurch, dass Methoden mit generischen Funktionen, deren Verhalten sie spezifizieren, und nicht mit Klassen assoziiert werden. Methoden und Methodenkombination werden in Kapitel 3.4 und 3.5 behandelt. Das abschliessende Kapitel 4 bietet einen Ausblick auf das Metaobject-Protokoll.

2 Dynamische vs. statische Typisierung

Lisp ist eine dynamisch getypte Sprache. Typen sind erst zur Laufzeit bekannt (d.h. die Typüberprüfung findet zur Laufzeit statt) und müssen nicht angegeben werden.. Variablen können Werte beliebigen Typs bezeichnen.
Dennoch ist festzuhalten, dass es sich bei Lisp um eine (wenngleich das nicht formal bewiesen wurde) typsichere Sprache handelt, d.h. nach Passieren des Typcheckers (zur Laufzeit) sind keine Typfehler mehr möglich.

Dynamische Typisierung bietet eine flexible Herangehensweise an die Programm-entwicklung, es können sehr schnell interaktiv Codeskizzen weiterentwickelt werden, doch sie bringt auch gravierende Nachteile mit sich:

- eine effiziente Speicherausnutzung wird erschwert, da der Speicherbedarf z.B. einer Variable zur Kompilierungszeit nicht bekannt ist
- es ergeben sich durch die Typüberprüfung zur Laufzeit Performancenachteile
- da in statisch getypten Sprachen bestimmte Fehlerklassen schon beim Kompilieren erkannt werden können, ergeben sich im Vergleich zu Lisp weniger Möglichkeiten für Typfehler während der Laufzeit
- Typdeklarationen erhöhen den Dokumentationswert des Codes

Letzterem wird entgegenet, indem der Programmierer Deklarationen für Parameter und/oder Rückgabewert angeben kann. Solch ein Beispiel vollständig deklarierten Codes sieht folgendermaßen aus:

```
(defun plus (x y)
   (declare (fixnum x y)
   (the fixnum (+ x y)))
```

Nun sollte ein Aufruf der Funktion **plus** mit einem Parameter falschen Typs zu einem Typfehler führen, ebenso ein Ergebnis der Addition, dessen Speicherbedarf grösser als der eines Fixnum ist. Denn Lisp führt zur Laufzeit nicht nur Typ- sondern auch Überlaufüberprüfung durch und bietet exakte Integer-Operationen. Im Normalfall würde bei einem Überlauf beispielsweise statt eines **fixnum** ein **bignum** alloziiert. Eine solche Überprüfung zieht natürlich zusätzliche Kosten nach sich.

Tatsächlich aber hängt das Verhalten des Compilers von diversen Voreinstellungen ab, die im Einzelnen sind: die Performance des generierten Codes (**speed**), der Umfang der Typüberprüfung zur Laufzeit (**safety**), der Speicherbedarf sowohl des Programmcodes als auch des Programms in Ausführung (**space**), die Debugginginformation, die mit dem Code gehalten wird (**debug**), und die Geschwindigkeit des Kompilierungsprozesses (**compilation-speed**). Jede dieser 5 Optionen kann mit einem Wert zwischen 0 und 3 gewichtet werden, wobei 0 die niedrigste und 3 die höchste Priorität hat.
Mit den unten gezeigten Optimierungen für speed und safety lässt sich Code generieren, der in Puncto Geschwindigkeit, aber auch in Puncto Sicherheit vergleichbar mit C-Code ist. Wie in C würde, falls bei der Addition tatsächlich ein Überlauf aufträte, stillschweigend das falsche Ergebnis zurückgeliefert. Das Setzen von **safety** auf Null kann sogar den Effekt haben, dass die Überprüfung auf die korrekte Anzahl der Übergabeparameter einer Funktion entfällt, was schwer zu lokalisierende Fehler zur Folge hat.

```
(defun plus (x y)
   (declare (fixnum x y)
            (optimize (speed 3) (safety 0)))
   (the fixnum (+ x y)))
```

Diese Optionen sollten daher sorgfältig und nicht global eingesetzt werden, sondern nur für Funktionen, die ausgereift genug sind, dass sich der Programmierer sicher sein kann, dass sie nur mit der korrekten Anzahl von Parametern aufgerufen werden kann und kein Überlauf auftritt. Zudem ist der Einsatz von Optimierung nur an Stellen sinnvoll und angebracht, wo ausgiebiges Profiling gezeigt hat, dass er tatsächlich einen Gewinn bringt.

3 Das Common Lisp Object System (CLOS)

Lisp verfolgt durchgehend die Philosophie, dass was gut für den Sprachdesigner ist, auch gut für den Programmierer ist. Wenn der Programmierer sich ein neues Feature wünscht, dass sein Programm einfacher zu schreiben macht, oder seine persönliche Programmieridee besser ausdrückt, bietet Lisp ihm die Möglichkeit, u.A. durch sein ausgereiftes Makrosystem, dieses Feature selbst zu integrieren. In anderen Sprachen wäre dazu eine langwierige und kostenintensive Änderung des Standards selbst nötig.

Erste Ansätze, objektorientierte Programmierung in Lisp zu integrieren gab es daher bereits Anfang der 80er Jahre, inspiriert von Smalltalk, eine Sprache, die Konzepte aus Simula, welches schon Features besaß, die der heutigen Auffassung von Klassen und Objekten sehr nahe kamen, mit traditionellen Ideen aus Lisp wie dynamischer Typisierung und Reflektion[1] verband.

[1] Die Fähigkeit eines Programmes, über seinen eigenen Programmcode zu räsonieren bzw. diesen auch abzuändern. Programmcode wird als Datum wie jedes andere betrachtet.

Wenige Jahre danach kam C++ auf den Markt, das aber, wie wir sehen werden, eine von Common Lisp grundsätzlich verschiedene Philosophie von Objektorientierung vertritt.

Zwei der frühen objektorientierten Ansätze waren New Flavors von Symbolics, Inc. und CommonLoops von Xerox PARC. Ab dem Sommer 1986 vereinten sich beide Gruppen, um deren Ansätze zu kombinieren und zu standardisieren. Als Ergebnis wurde das sog. *Common Lisp Object System* (CLOS) bei der Standardisierung Common Lisps 1994 Teil des ANSI-Standards. In der Tat war Common Lisp sogar die erste objektorientierte Programmiersprache, die einen ANSI Standard erhielt.

3.1 Integration

CLOS ist in Common Lisp voll integriert und es macht wenig Sinn, es als separate Einheit zu betrachten. Das Wort Objekt bezeichnet jedes beliebige Lisp Datum, Common Lisp unterscheidet nicht wie manche Sprachen zwischen Klassen und primitiven Datentypen; alle Daten in Common Lisp sind Objekte und jedes Objekt ist Instanz einer Klasse.

Die primitiven Datentypen in Lisp sind:

- symbol
- boolean*
- cons
- numerische Typen:
  - bit*
  - fixnum*
  - bignum*
  - float[2]
  - rational
  - complex
- character
- Container:
  - string
  - array (Elementtypen können variieren)
  - list (Elementtypen können variieren)
  - hash-table
- function (inclusive Funktionen höherer Ordnung als first-class Objekte)
- path-name

CLOS verlangt es, dass alle Klassen einen korrespondieren Typ besitzen. Jedoch muss nicht für jeden „traditionellen" Lisp-Typ eine Klasse existieren. Solche Ausnahmen sind die mit * gekennzeichneten Typen, die implementierungsabhängig vorhanden sein können, aber nicht vom Standard vorgeschrieben sind. So kann 1 z.B. den Typ **bit** haben, aber Instanz der Klasse **integer** sein. Die Klassenhierarchie der sog. *eingebauten (built-in) Klassen* zeigt Abb. 1.

Diese eingebauten Klassen können jedoch weder instanziiert werden noch als Superklassen in einer Klassendefinition angegeben werden. Jedoch können auf sie Methoden definiert werden, die ihr Verhalten erweitern (siehe Kap. 3.4)

[2] Vom Standard vorgeschrieben sind IEEE 64-bit Fließkommazahlen, einige Implementierungen, wie GNU Common Lisp bieten aber sogar beliebig genaue Fließkommazahlen an.

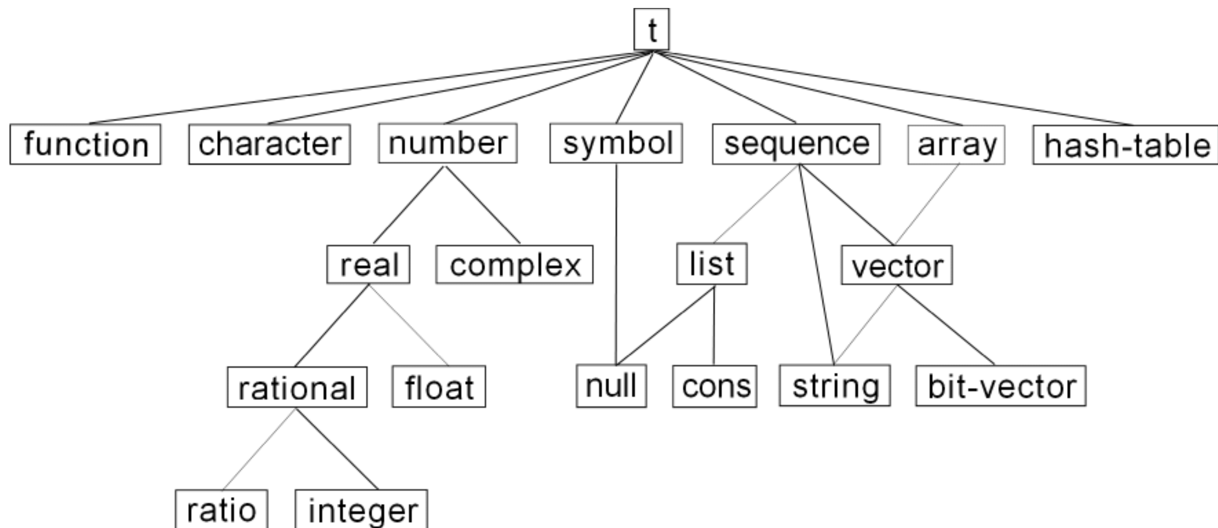

Abb 1: Eingebaute Klassen

Auch das aus Rückwärtskompatibilität integrierte **defstruct**, das Typeltypen einführt**,** wurde so angepasst, dass es Klassen definiert, die jedoch den selben Einschränkungen wie eingebaute Klaasen unterliegen.

Im Gegenstatz dazu stehen die sog. *Standard-Klassen*. Sie sind Subklassen von **standard-object**. Die Klassendefinitionen **(defclass foo () ())** und **(defclass foo (standard-object) ())** sind daher völlig äquivalent (siehe Kap. 3.2).
Standard-object selbst ist wiederum Subklasse von **t**[3] . Betrachtet man die Abhängigkeiten von Klassen als Baum, so repräsentiert **t** dessen Wurzel und ist somit direkte oder indirekte Superklasse aller Klassen. Jedes Datum in Common Lisp ist Instanz der Klasse **t**, da jede Instanz einer Klasse gleichzeitig auch Instanz aller ihrer Superklassen ist[4.] Mithin sind alle Klassen in Common Lisp Teil einer einzigen Klassenhierarchie.

3.2 Klassen:

Wie wir bereits gesehen haben ist Common Lisp wie die meisten heutigen objektorientierten Sprachen klassenbasiert. Eine Klasse kann definiert werden als Subklasse anderer Klassen, ihrer Superklassen. Objekte werden durch Instanziierung erzeugt. Eine Klasse ererbt Teile ihrer Definition und ihres Verhaltens von ihren Superklassen und Instanzen einer Klasse sind gleichzeitig Instanzen der Superklassen.[5] Vererbung ist transitiv. Eine Klasse kann weder direkte noch indirekte Subklasse ihrer selbst sein. Somit sind Klassen in einem gerichteten azyklischen Graphen, dem *Klassengraphen* organisiert. Abb. 2 zeigt einen Ausschnitt aus einem solchen Graphen am Beispiel einer Klasse **musik-instrument**.

Ein Musikinstrument kann sein ein Schlaginstrument wie eine Trommel, ein Saiteninstrument, ein Tasteninstrument wie ein Klavier oder ein Blasinstrument wie eine Trompete. Ein Saiteninstrument besitzt die Subklassen **streich-instrument**, von dem eine Klasse **geige** erbt, und **zupf-instrument**, von dem etwa eine Klasse **gitarre** erben könnte.

[3] nicht zu verwechseln mit dem Symbol **t**, das Instanz der Klasse Symbol und damit nur indirekt der Klasse **t** ist.
[4]Im Gegensatz dazu stehen prototypen-basierte objektorientierte Sprachen wie z.B. JavaScript, in denen Objekte durch Klonen eines Prototypen-Objektes erzeugt werden. Der Klon kann im Folgenden modifiziert und als Prototyp für andere Objekte benutzt werden.

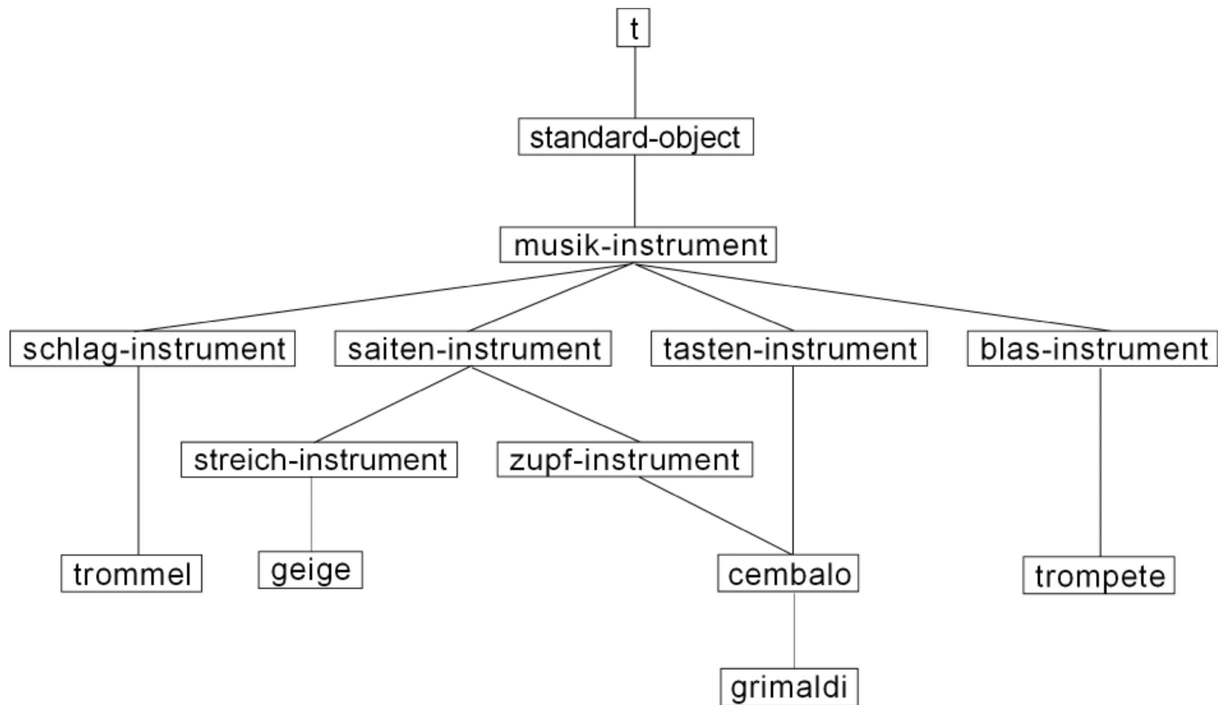

Abb 2. Standard-Klassen

CLOS unterstützt multiple Vererbung, d.h. eine Klasse kann mehrere direkte Superklassen haben, d.h. Klassen, die aber wiederum selbst nicht in einem Sub-/Superklassenverhältnis stehen. Ein Beispiel sehen wir in Abb.2: Ein Cembalo besitzt offensichtlich Tasten und Saiten, die aber mittels einer Mechanik angezupft werden. Somit erbt die Klasse **cembalo** sowohl von **zupf-instrument** als auch von **saiten-instrument**.

Eine Klasse erbt nicht nur von direkten sondern auch von indirekten Superklassen. Dabei sind letztere weniger *spezifisch* als erstere, ebenso wie eine Klasse spezifischer ist als deren direkte Superklasse(n). Weiterhin sind per Definition Superklassen, die in der Klassendefinition (s. U.) vor einer anderen Klasse aufgelistet sind, spezifischer als nachfolgend aufgeführte. Das mag beliebig wirken, ermöglicht jedoch eine totale Ordnung auf Klassen, und damit die Erstellung einer sog. *Klassenpräzedenzliste,* die im Folgenden von Bedeutung sein wird.
Ein Beispiel: Vorausgesetzt, **zupf-instrument** wird in der Klassendefinition als Superklasse vor **saiten-instrument** aufgelistet, ergibt sich für **cembalo** folgende Klassenpräzedenzliste:
(**cembalo**, **zupf-instrument**, **saiten-instrument**, **tasten-instrument**, **musik-instrument**, **standard-object**, **T**)

Betrachten wir nun das Makro **defclass,** das eine neue Standard-Klasse definiert. Seine Struktur sieht folgendermassen aus:
defclass *class-name ({superclass-name}*) ({slot-specifier}*) [[class-option]]*
Eine Klassendefinition spezifiziert neben den direkten Superklassen und den Klassenoptionen, die sich jeweils auf die *gesamte* Klasse beziehen und etwa die Erstellung einheitlicher **:accessor**-Methoden oder eines Konstruktors ermöglichen, außerdem die S*lots* (in anderen objektorientierten Sprachen *Felder*, *Instanzvariablen*, oder *Attribute*), aus denen sich Instanzen zusammensetzen.
Ein *Slot-Specifier* besteht aus einem *Slot-Namen* und einer Reihe von *Slot-Optionen*. Diese sind im einzelnen **:reader**, **:accessor, :initform**, **:initarg** und **:allocation**.

Durch Angabe einer **:reader**-Option wird eine Methode zum Lesen des slots generiert, analog dazu generiert **:accessor** eine Methode zum Lesen und Schreiben.
:initarg erlaubt die Angabe eines benannten Initialisierungsargumentes für einen Slot bei der Instanziierung.
Man beachte, dass **:reader-**, **:accessor-,** und **:initform-**Argumente gleichlautend mit dem Slot-Namen sein können, es aber nicht sein müssen.
Weiterhin ist die Angabe eines Initialisierungsargumentes (**:initarg**) möglich, das ausgewertet wird, sofern kein **:initform-**Argument explizit spezifiziert wurde.
Es gibt Slots, die jede Instanz für sich besitzt (der Slot-Specifier **:allocation** wird auf **:instance** gesetzt), und solche, die nur einmal für eine ganze Klasse (**:allocation :class**) angelegt werden.
Betrachten wir als Beispiel die Klasse **grimaldi** aus Abb. 2. Da es sich um sehr kostbare Instrumente, gebaut im 17. Jahrhundert in Italien, handelt, möchten wir den gesamten Weltbestand erfassen. Offensichtlich reicht es völlig aus, dies an einer einzigen, für alle Instanzen zugänglichen Stelle zu tun. Findet sich nun zufällig irgendwo auf der Welt ein bisher unbekanntes Instrument, so müssen wir nur ein einziges Mal den Slot-Wert für **exemplare** inkrementieren anstatt die Änderung in jeder einzelnen Instanz separat vorzunehmen.

```
(defclass tasten-instrument (musik-instrument)
   ((oktaven :reader oktaven : initarg :oktaven)
    (hersteller :reader fabrikant :initarg :fabrikant
                 :initform „kein fabrikant angegeben“)))

(defclass saiten-instrument (musik-instrument)
   ((saiten :reader saiten :initarg :saiten)
    (hersteller :reader erbauer :initarg :erbauer
                 :initform „kein erbauer angegeben“)))

(defclass cembalo(zupf-instrument tasteninstrument)())

(defclass grimaldi (cembalo)
   ((hersteller :initform “Grimaldi“)
    (exemplare :accessor exemplare :allocation :class)))
```

Der Nutzen von **:accessor**-Methoden für die Erweiterbarkeit von Programmen liegt auf der Hand. Änderungen müssen nur an einer Stelle des Programms vorgenommen werden und nicht überall dort, wo mit **slot-value** auf einen Slot zugegriffen wird. Etwa könnte eine **:reader**-Methode nun ohne Probleme derart neu definiert werden, dass sie nicht mehr lediglich einen Slot-Wert zurückzugibt, sondern den Wert aus mehreren slot-Werten jeweils neu berechnet.

Es liegt allerdings in der Verantwortung des Programmierers, **:reader**- und **:accessor**-Methoden auch tatsächlich zu benutzen. Die Instanzvariablen können weiterhin mit **slot-value** (auf dessen Basis die **:reader**- und **:accessor**-Methoden im übrigen implementiert sind), abgefragt und mit setf geschriebenden werden. Eine Unterscheidung - wie etwa in Java - zwischen *public – private* Slots existiert nicht. Das CLOS *zielt nicht darauf* ab, Probleme von Verkapselung und Sicherheit zu lösen, da dies konträr zur von Lisp vertretenen Philosophie der Reflexivität stünde.

Jede Klasse besitzt die Slots, die in der Klassendefinition angegeben wurden plus diejenigen, die von den (direkten sowie indirekten) Superklassen geerbt werden.
Dabei kann ein gegebenes Objekt nur jeweils einen Slot mit einem bestimmten Namen haben. Jedoch ist es möglich, dass mehr als eine Klasse in der Vererbungshierarchie einen Slot mit diesem Namen spezifizieren.
Das kann geschehen entweder wenn eine Subklasse einen Slot einer Superklasse neu definiert, oder wenn zwei oder mehrere Superklassen Slots gleichen Namens haben. In solchen Fällen müssen Strategien gefunden werden um mehrere Slots zu einem zu vereinigen.

Dabei werden **:initform**-Werte von der in der Klassenpräzedenzliste spezifischsten Klasse übernommen und können also von einer Subklasse überschrieben werden.
:initarg-Argumente müssen dagegen keine einheitliche Repräsentation haben und werden daher schlichtweg vereinigt. Selbiges gilt für **:reader**- und **:accessor**-Methoden.
:allocation ist bei der (Re-)Definition eines slots per default stets auf **:instance** gesetzt.

Wie von einer dynamisch getypten Sprache zu erwarten, können Klassen zur Laufzeit abgeändert werden oder sogar Instanzen die Klasse wechseln. Im ersteren Fall wird die Veränderung automatisch an die Instanzen weiterpropagiert.
Der zweite Fall verdient eine nähere Betrachtung. Nehmen wir an, wir verfügen über eine Klasse **position** mit zwei Unterklassen **x-y-position** und **x-y-z-position**:

```
(defclass position () ())

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

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

Nehmen wir weiterhin an, wir haben bereits eine Instanz der Klasse **x-y-position** erzeugt:

```
(setf my-position (make-instance `x-y-position :x 3 :y 5))
```

Zu spät bemerken wir nun, dass wir für **my-position** doch die Dreidimensionalität benötigen. Dem kann folgendermassen abgeholfen werden:

```
(change-class my-position 'x-y-z-position :z  0)
```

Die x- und y- Werte der jeweiligen Slots bleiben auch nach der Klassenänderung erhalten, da die Namen alter wie neuer Slots gleichlautend sind.

3.3 Generische Funktionen

CLOS stützt sich im Gegensatz zum sog. *Message-Passing*, wie es in Simula eingeführt wurde und wie man es heute in vielen anderen objektorientierten Programmiersprachen wie C++ oder Java findet, auf *generische Funktionen*, d.h. Funktionen, die Parameter unterschiedlicher Typen akzeptieren, und deren Verhalten von den Klassen dieser Parameter abhängt.

Betrachten wir ein Beispiel: Wir möchten zu einem gegebenen Musikinstrument seinen Klang ausgeben lassen und instanziieren dazu ein Objekt der Klasse **musik-instrument**, genauer gesagt deren (indirekter) Subklasse **trompete** (siehe Abb. 2). In Java würde der entsprechende Code zur Klangausgabe etwa folgendermassen aussehen:

t.klang();

Was dabei geschieht ist Folgendes: Dem Objekt t wird eine „Nachricht“ (Message), in diesem Fall ohne Parameter, gesandt, worauf dieses den auszuführenden Code der Methode **klang()** anhand der Definition seiner Klasse **trompete** ausfindig macht. Da jede Klasse ihre eigene Methode eines gegebenen Namens besitzen kann, führt dieselbe Nachricht, an verschiedene Objekte (z.B der Klasse **piano**, **geige**, etc.) gesandt, zur Ausführung unterschiedlichen Codes.
In CLOS sind jedoch Methoden nicht mit Klassen assoziiert, sondern mit sog. *generischen Funktionen*. Natürlich könnte man das Message-Passing in obigem einfachen Fall nachbilden, indem man eine spezielle Funktion **send** einführt.
Unser obiges Beispiel würde dann in Lisp-Code wie folgt aussehen:

(send t 'klang)

Tatsächlich wurde in frühen Lisp-Objektsystemen genau so gearbeitet. Diese Herangehensweise war jedoch aus mehreren Gründen unbefriedigend. Zum einen unterschied sich ein solcher Methodenaufruf syntaktisch von einem normalen Funktionsaufruf, der so aussehen würde (Und in CLOS auch genau so aussieht):

(klang t)

Schlimmer noch, dadurch, dass Methoden keine Funktionen sind, ging die Konsistenz zur funktionalen Programmierung verloren. Um unsere Methode einer Funktion höherer Ordnung wie **mapcar** zu übergeben, damit sie auf alle Elemente einer Liste angewandt werden kann, hätte man (umständlich) schreiben müssen:

(mapcar #'(lambda (instrument) (send instrument 'klang)) instrument-liste)

anstatt von:

(mapcar #'klang instrument-liste)

Denken wir einen Schritt weiter: Was wäre, wenn wir den Klang einer Trommel ausgeben möchten? Offensichtlich hängt dieser nicht nur von der Art der Trommel (eine Snare klingt anders als eine Bassdrum, und diese wieder anders als eine Konga), sondern auch von der Art des Schlagwerkzeugs (Hand, Stock oder Bürste) ab. In einem traditionell Message-Passing System stellt sich die Frage, welcher der beiden Klassen **trommel** oder **schlagwerkzeug** unsere Methode **klang** zuzuordnen wäre. Eigentlich beiden!

Der Ansatz, generische Funktionen zu verwenden, bietet hier in konzeptioneller Einfachheit und Klarheit Ausdrucksmöglichkeiten, die mit Message-Passing nur unter grossem Aufwand zu simulieren sind. Denn wir rufen schlichtweg eine generische Funktion mit *beiden* Objekten, dem Trommel-Objekt *und* dem Schlagwerkzeug-Objekt auf:

(klang t s)

Dies sieht nicht nur wie ein normaler Funktionsaufruf aus, sondern generische Funktionen sind tatsächlich First-Class-Objekte und können wie normale, mit **defun** definierte Funktionen in Funktionsaufrufen übergeben und als Resultate zurückgeliefert werden.
Betrachten wir nun die eigentliche Definition einer generischen Funktion. Sie hat für unser Beispiel folgende Form:

```
(defgeneric klang (instrument werkzeug)
    (:documentation "Gibt den Klang eines Instruments abhängig vom verwendeten Werkzeug aus."))
```

Auffällig ist allerdings, dass diese Definition einer generischen Funktion im Vergleich zu einem normalen **defun** keinen Funktionsrumpf enthält. Generische Funktionen definieren lediglich eine abstrakte Operation, die zwar einen Namen und eine Parameterliste (und eine **:documentation**-Option, deren Benutzung besonders in komplexeren Fällen anzuraten ist) besitzt, aber keine Implementierung. Mit Inhalt gefüllt werden generische Funktionen erst durch Methoden.

3.4 Methoden

Eine Methode bietet die Implementierung einer generischen Funktion für bestimmte Klassen von Argumenten. Die Zuordnung von Methoden zu generischen Funktionen erfolgt durch Namensgleichheit. Möchte man die gegebene generische Funktion **klang** aus dem Beispiel des vorigen Kapitels um eine neue Methode für ein spezifisches Instrument erweitern, so muss deren Name ebenfalls **klang** lauten. Diese Methode kann wie auch normale Funktionen und analog zu Klassen abgeändert, oder überschrieben werden durch erneute Definition einer Methode selben Namens mit gleicher Parameterliste.

Bei erstmaliger Definition einer Methode wird zwar, falls nicht existent, implizit eine korrespondierende generische Funktion generiert, jedoch empfiehlt es sich aus Dokumentationsgründen (siehe voriges Kapitel), diese stets explizit zu definieren.

Weiterhin ist zu beachten, dass die Parameterliste einer Methode kongruent sein muss zu der ihrer generischen Funktion, d.h sie muss dieselbe Anzahl von erforderlichen und optionalen Parametern akzeptieren, sowie alle Argumente, die eventuellen **&rest** oder **&key** Parametern entsprechen[5]. Eine Methode kann **&key**- und **&rest**-Argumente der generischen Funktion akzeptieren, einfach indem der Lambda-Liste ein **&rest**-Parameter angehängt wird, indem sie identische **&key**-parameter spezifiziert, oder durch **&allow-other-keys** in Verbindung mit **&key**. Weiterhin kann eine Methode auch eigene, in der Definition der generischen Funktion nicht vorkommende **&key** Parameter angeben. Wenn die generische Funktion aufgerufen wird, werden automatisch auch diese Parameter akzeptiert.
Die eigentliche Selektion der auszuführenden Methode(n) (*Dispatching*) zu den ihr übergebenen Parametern übernimmt die generische Funktion basierend auf der Struktur des Klassengraphen. Wie dies vor sich geht, wird im Folgenden beschrieben.
Hierzu betrachten wir zunächst das Makro **defmethod**. Wie unten zu sehen, unterscheidet es sich von **defun** nur in einem einzigen, aber wichtigen Punkt. Jeder erforderliche Parameter der Lambda-Liste kann (muss aber nicht) ersetzt werden durch einen Ausdruck der Form (**Parameter Specializer**). Ein Parameter kann auf zwei Arten spezialisiert werden. Der normale Fall wäre die Angabe einer Klasse deren (direkte oder indirekte) Instanz das Argument sein muss. Wird kein expliziter Specializer angegeben, so ist dies gleichbedeutend mit einer Spe-

zialisierung auf **t**. Die zweite Art von Specializer ist der sog. **eql**-*Specializer*, der explizit ein bestimmtes Objekt angibt, auf das die Methode angewandt werden soll.

(defmethod test-int ((n number)) (print “num”))

(defmethod test-int ((i integer)) (print “int”))

(defmethod test-int ((i (eql 0))) (print “null”))

(test-int 1/2)	**=>**	**“num“**
(test-int 1)	**=>**	**“int“**
(test-int 0)	**=>**	**„null“**

Hier sieht man, dass eine Methode gleichnamige Methoden, die auf ihre Superklassen spezialisieren, verschattet. Sie ist *spezifischer* als die Methoden der Superklassen, da auch die Klasse, auf die sie spezialisiert, laut Klassenpräzedenzliste spezifischer ist als die Klassen, von denen sie erbt (siehe Kapitel 3.2). **Eql**-Specializer sind immer spezifischer als Klassen-Specializer.
An obigem Beispiel erkennt man nebenbei bemerkt auch, dass nicht nur auf benutzerdefinierte Klassen spezialisiert werden kann, da auch die meisten built-in-Typen durch Klassen repräsentiert werden (vergleiche Kapitel 3.1)
Interessanter wird der Fall, wenn Methoden auf mehr als eine Klasse spezialisieren (sog. *Multimethoden*). Um dann zwei gegebene anwendbare Methoden zu ordnen, vergleicht die generische Funktion ihre Parameter-Specializer von links nach rechts und der erste abweichende Specializer bestimmt die Ordnung, wobei wieder der spezifischste Specializer Vorrang hat. Ein weiteres Beispiel hierzu:

(defmethod test-int2 ((x number) (y number)) (print “num num”))

(defmethod test-int2 ((i integer) (y number)) (print “int num”))

(defmethod test-int2 ((x number) (j integer)) (print “num num”))

[5] &rest Parameter erlauben einer Funktion eine variable Anzahl von Parametern zu empfangen. Ein Beispiel ist die Funktion +:

(+)	=>	0
(+ 1)	=>	1
(+ 1 2)	=>	3
(+ 1 2 3)	=>	6

Eine Funktion kann erforderliche und optionale Parameter besitzen. Ein Problem ergibt sich jedoch, wenn ein Benutzer beispielsweise nur einen Wert für den dritten optionalen Parameter angeben möchte. **&key** Parameter lösen dieses Problem, indem ein Argument explizit unabhängig von der Position in der Parameterliste angeben werden kann.

(test-int2 1/2 1/2) => **“num num”**

(test-int2 1/2 1) => **“num int”**

(test-int2 1 1/2) => **“int num”**

(test-int2 1 1) => **“int num”** ***;* nicht “num int”!**

Zusammengefasst läuft der Dispatch-Mechanismus beim Aufruf einer generischen Funktion bisher also folgendermaßen ab:

1. Berechnen einer Liste von *anwendbaren Methoden;*
2. Ist keine Methode anwendbar, so wird ein Fehler signalisiert;
3. Sortieren der anwendbaren Methoden in der Reihenfolge ihrer Spezifizität;
4. Aufruf der speziellsten Methode.

3.5 Methodenkombination

Beim Dispatching bleiben die weniger spezifischen anwendbaren Methoden weiter erreichbar, und zwar durch Aufruf der lokalen Funktion **call-next-method** am Ende des Methodenrumpfes. Diese Funktion übergibt die Kontrolle an die jeweils nächstspezifische Methode. Den endgültigen Rückgabewert bestimmt die zuletzt aufgerufene Methode.
call-next-method kann auf zwei Arten aufgerufen werden:

- Ohne Argumente; in diesem Fall bekommt die next-Methode exakt dieselben Parameter wie die aufrufende, oder
- Mit expliziten Argumenten. Dann muss aber sichergestellt bleiben, dass sich die Liste anwendbarer Methoden dadurch nicht verändert.

Ein Aufruf von **call-next-method** führt zu einem Fehler, wenn keine Next-Methode existiert. Jedoch ermöglicht es eine weitere lokale Funktion, **next-method-p**, herauszufinden, ob es eine solche Next-Methode gibt. Beide Funktionen haben keine Bedeutung ausserhalb eines Methodenrumpfes (daher die Bezeichnung lokal).

Zusätzlich zu den bisher beschriebenen sog. *Primärmethoden* gibt jedoch auch *Hilfsmethoden*, die **:before**-, **:after**-, und **:around**-Methoden. Die Definition einer Hilfsmethode erfolgt analog zu der einer Primärmethode durch das **defmethod**-Makro, mit dem Unterschied, dass zwischen Methodennamen und Parameterliste der sog. *Methoden-Qualifier* auftaucht. Definieren wir uns zur Anschauung eine **:before**-Methode zu unserer generischen Funktion **test-int**. Diese Definition könnte wie folgt aussehen:

(defmethod test-int :before ((n number))
(print “Diese Methode testet, ob eine Zahl eine Ganzzahl gleich oder ungleich Null ist“))

Beim Aufruf einer generischen Funktion werden alle ihre anwendbaren Primär- und Hilfsmethoden zu einer einzigen sog. *effektiven Methode* kombiniert. Dabei werden, wie die Namen suggerieren, **:before**-Methoden vor allen Primärmethoden ausgeführt und können mithin dazu dienen, für deren Ausführung nötige vorbereitende Schritte zu unternehmen. Analog dazu werden **:after**-Methoden nach allen Primärmethoden aufgerufen und übernehmen sozusagen die „Aufräumarbeit“. **:around**-Methoden werden um die **:before**-,

Primär- und **:after**-Methoden herum ausgeführt. Der Begriff „around" stammt daher, dass **call-next-method** auch in der Mitte eines Methodenrumpfes aufgerufen werden kann, so dass ein Teil des Codes vor allen anderen **:around-**, **:before**-, Primär- und **:after**-Methoden und der Rest des Codes nach denselben. Selbiges ist im Übrigen auch bei Primärmethoden möglich. Typischerweise werden **:around**-Methoden benutzt, um irgendeine Art von dynamischem Kontext zu etablieren, etwa eine dynamische Variable zu binden oder einen Error-Handler einzuführen [6].

Im Einzelnen geht die Bildung der effektiven Methode folgendermaßen von Statten:

1. Berechnen der anwendbaren Methoden, und Aufteilen dieser gefundenen Methoden in separate Listen anhand ihrer Qualifier;
2. Existiert keine anwendbare Primärmethode, so wird ein Fehler signalisiert.
3. Sortieren jeder der in 1. gebildeten Listen nach Spezifität der Methoden;
4. Ausführen der spezifischsten **:around**-Methode und Rückgabe deren Resultatwerts;
5. Falls eine **:around**-Methode **call-next-method** aufruft, wird die nächste weniger spezifische **:around**-Methode ausgeführt;
6. Wenn von Vornherein gar keine **:around**-Methoden definiert wurden oder wenn eine **:around**-Methode **call-next-method** aufruft, ohne dass weitere **:around**-Methoden vorhanden sind, wird wie folgt vorgegangen:
    a. Ausführen *aller* **:before**-Methoden in der Reihenfolge ihrer Spezifizität. Rückgabewerte werden ignoriert. Aufrufe von call-next-method oder next-method-p sind nicht erlaubt;
    b. Ausführen der spezifischsten Primärmethode;
    c. Wenn eine Primärmethode **call-next-method** aufruft, wird die nächstspezifischste Primärmethode zur Ausführung gebracht;
    d. Existiert keine solche, so wird ein Fehler signalisiert;
    e. Rückgabe des Resultatwerts der am wenigsten spezifischsten Primärmethode, falls keine **:around**-Methode vorhanden war;
    f. Nach Abarbeitung aller vorhandenen Primärmethoden werden alle **:after**- Methoden, jedoch in *umgekehrter Reihenfolge* ihrer Spezifizität ausgeführt, wobei wieder alle Rückgabewerte ignoriert werden und wie bei **:before**-Methoden Aufrufe von **call-next-method** oder **next-method-p** verboten sind.

Das Berechnen, Sortieren und Ausführen einer Liste anwendbarer Methoden sind aufwändige Operationen. Typischerweise wird eine Implementierung daher die effektive Methode cachen, da es wahrscheinlich ist, dass der nächste Aufruf der generischen Funktion wieder Parameter selben Typs hat. Wird eine neue Methode auf der generischen Funktion definiert, so ist es meist günstiger, den Cache zu verwerfen, als zu versuchen, ihn zu modifizieren. Dies stellt jedoch kein allzu großes Problem dar, da Methodendefinitionen viel seltener vorkommen dürften als Methodenaufrufe.

Zusätzlich zu dieser bisher eingeführten, *Standard-Methodenkombination* genannten Vorgehensweise, gibt es noch die *einfachen eingebauten Kombinationen*, die Methoden durch alternative Operationen wie Addition, Konjunktion oder Disjunktion, Listenbildung oder Bestimmung des Maximums oder Minimums der einzelnen Methoden verknüpfen. Diese sind: +, **AND**, **OR**, **LIST**, **APPEND**, **NCONC**, **MIN**, **MAX**, und **PROGN.** Diese einfachen Methodenkombinationen unterstützen keine **:before**- und **:after**-Methoden.

[6] Es lässt sich zwar mit Hilfsmethoden nichts ausdrücken, was sich nicht auch durch Primärmethoden nachbilden ließe, dennoch bieten sie konzeptionelle Klarheit und Eleganz, indem sie die Intention des Programmierers wiederspiegeln.

Es ist darüber hinaus möglich, eigene Methodenkombinationen zu konstruieren, auch wenn sich diese Notwendigkeit in der Praxis nur äußerst selten ergeben dürfte.

Ähnliche Vorgehensweisen finden sich in der aspektorientierten Programmierung, welche aufgrund der Einsicht entwickelt wurde, dass einige übergreifende programmiererische Anliegen wie Logging (die Protokollierung des Programmablaufs in sog. Logdateien), Fehlerbehandlung oder Persistenz sich schlecht einzelnen Klassen zugeordnet werden können. Diese mussten bisher verstreut an vielen Stellen des Programmcodes separat implementiert werden. Mit Hilfe der aspektorientierten Programmierung können diese sog. *Aspekte* in einzelne Module gekapselt werden und später in den Programmcode eingeflochten werden, was die Wartbarkeit und Wiederverwendbarkeit enorm verbessert. Hierzu werden an bestimmten Stellen des Codes Punkte gesetzt, z.B bei der Ausführung von Methoden. Sogenannte **before-**, **after-** und **around**-**Advices** bestimmen dann, wo der einzuflechtende Code eingebracht wird.
Von der Intention her ist diese Technik vergleichbar mit dem CLOS-Metaobjekt-Protokoll, das im nächsten Kapitel beschrieben wird; nicht zufällig wurden beide von Xerox Parc entwickelt.

4 Das Metaobjekt-Protokoll

Das Metaobjekt-Protokoll erweitert die CLOS ANSI-Spezifikation, indem es CLOS selbst als erweiterbares CLOS-Programm auffasst. Es ist jedoch lediglich ein de-facto-Standard. Das Metaobjekt-Protokoll beruht darauf, dass Klassen first-class Objekte sind und damit selbst wieder Instanzen von Klassen, den sog. Metaklassen. Alle Metaklassen sind Subklassen von **metaobject** (siehe Abb 3). Grau hinterlegte Klassen sind abstrakte Klassen, die nicht zur Instanziierung gedacht sind. Das Resultat des Versuchs, eine solche Klasse zu instanziieren, ist undefiniert.

Für jedes Programmelement existiert eine sog. *Basismetaobjektklasse*. Diese sind im Einzelnen: **generic-function**, **class**, **slot-definition**, **method** und **method-combination**. Eine Metaobjektklasse ist eine Subklasse von exakt einer Subklasse einer dieser Klassen. Wieder ist das Resultat eines Versuchs, eine Klasse zu definieren, die Subklasse von mehr als einer Basismetaobjektklass ist, undefiniert. Ein Metaobjekt ist eine Instanz einer Metaobjektklasse. Insbesondere sind alle Klassen, die wir bisher kennengelernt haben, Metaobjekte.

Mit jedem Metaobjekt assoziiert ist die Information, die es benötigt, um seine Rolle zu erfüllen. Diese Informationen können direkt aus einem **defclass-** oder **defmethod-**Macro oder aber indirekt etwa aus der Klassenhierarchie gewonnen werden und sind immer basiert auf die anderer Metaobjekte. So gehört zu einer generischen Funktion die Menge aller Methoden, die auf ihr definiert wurden. Ein **class**-Metaobjekt beinhaltet die direkten Super- und Subklassen sowie die Klassenpräzedenzliste ebenfalls als **class**-Metaobjekte und die Methoden, die die Klasse als Specializer benutzen zusammen mit den zugehörigen generischen Funktionen als **method**-bzw **generic-function**-Metaobjekte. Weiterhin gehören zu ihr ihre **slot-definitions,** sowohl die direkt in der Klasse definierten, als auch die ererbten. Ein **slot-definition**-Metaobjekt spezifiziert die generischen Funktionen für die automatisch generierte **reader-** und **writer**-Methoden existieren. Ein **method**-Metaobjekt hält die zugehörige generische Funktion.
Durch ein **defgeneric**-Macro wird **standard-generic-function** instanziiert, durch **defclass standard-class** und durch **defmethod standard-method**.

Metaklassen selbst sind ganz normale Klassen und somit Instanz der Klasse **standard-class**. **Standard-class** ist wiederum Instanz von sich selbst. Dies ist die fundamentale Zirkularität

des Metaobjekt-Protokolls.
Von Metaklassen können wie von jeder anderen Klasse Subklassen mit erweitertem Verhalten definiert werden. So kann das Verhalten von CLOS-Klassen an neue Anforderungen, wie zum Beispiel Persistenz, angepasst werden.

Schlussendlich ist es mit Hilfe des Metaobjekt-Protokoll möglich, auch eine komplett neue objektorientierte Sprache zu implementieren.

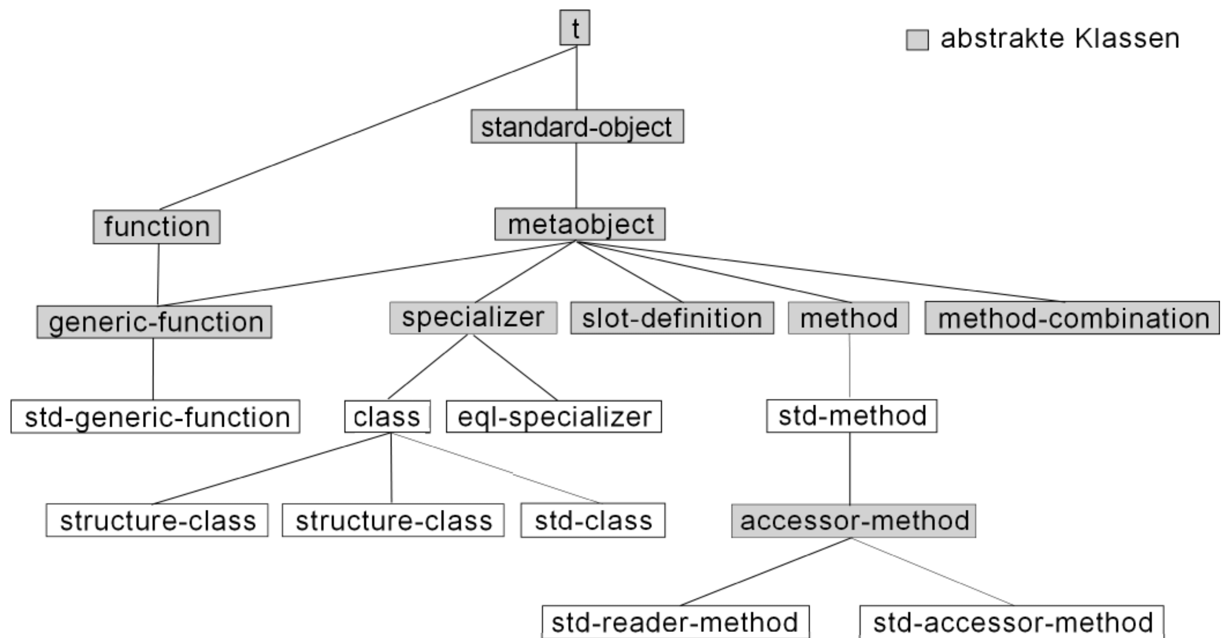

Abb. 3: Metaklassen

5 Quellen

[1] Peter Seibel. **Practical Common Lisp**, Kapitel 16 und 17. Apress, April 2005

[2] Linda G. DeMichiel and Richard P. Gabriel. **The Common Lisp Object System: An overview**. In J. Bezivin et al., editor, *ECOOP '87, European Conference on ObjectOriented Programming*, volume 276 of *LNCS*. Springer, 1987

[3] Richard Gabriel, Jon White, and Daniel Bobrow. **CLOS: Integrating objectoriented and functional programming.** CACM: Communications of the ACM, 34, 1991.

[4] Sonya E. Keene: **Object-Oriented Programming in Common Lisp.** Addison Wesley. 1989

[5] G. Kiczales, J. des Rivieres, and DG Bobrow. **The Art of the Metaobject Protocol**. MIT Press, 1991

[6] http://www.lisp.org/mop/

[7] http://p-cos.net/lisp/guide.html