Objektorientierte Programmierung mit COMMON LISP / C L O S

CLOS [2] ist das Akronym für COMMON LISP Object System. Es stellt eine Spracherweiterung der zwischenzeitlich weitestgehend standardisierten Programiersprache COMMON LISP [5] dar. CLOS ist eine Menge von Operatoren für objektorientiertes Programmieren und bildet seit 1988 einen offiziellen Spracherweiterungsstandard von COMMON LISP zur objektorientierten Programmierung, wodurch hohe Portierbarkeit von in COMMON LISP/CLOS geschriebenen Programmen erreichbar wird. Anhand eines Beispiels aus [1] (siehe Abschnitt 2) wird die traditionelle (funktionale) Lösung[1] eines kleinen Problems einem objektorientierten Ansatz zur Veranschaulichung gegenübergestellt. Dass auch oder gerade komplexe Projekte in COMMON LISP und CLOS implementierbar sind, zeigen etwa die in [3] beschriebenen Programmierbeispiele von Experten-(Konstruktions-)Systemen.

Dieser CLOS-Artikel umreißt die Grundideen der objektorientierten Programmierung und zeigt die wesentlichen Bestandteile der Spracherweiterung CLOS gemäß der Beschreibung in [2] auf. Dem interessierten Leser möge diese Zusammenstellung als Auffrischung seiner bereits bestehenden CLOS-Kenntnisse oder aber als kurze Einführung in diese Thematik bzw. als Übersicht dienen.

Inhaltsverzeichnis:

[1] In [6] wird hierbei auch vom ausführungsorientierten Ansatz gesprochen.

1 Grundideen der objektorientierten Programmierung

Objektorientierung ist allgemein ein Hilfsmittel zur gedanklichen und softwaretechnischen Strukturierung und Modellierung, welches dazu beiträgt, die sog. "semantic gap", die semantische Differenz zwischen Beschreibungsmittel und realem Kontext einer Problemstellung, zu minimieren. Der objektorientierte Entwurf ist kein Novum, er basiert im Wesentlichen auf dem Begriff des abstrakten Datentyps.

Paradigmenwechsel gemäß [1]: Im Gegensatz zur konventionellen Programmierung, bei der Programme bestimmte Daten bearbeiten, wissen bei der objektorientierten Programmierung die Daten(-objekte) selbst, wie sie sich zu verhalten haben, und Programme bestehen aus Interaktionen zwischen diesen Daten(-objekten). Objektorientierte Programmierung bedeutet eine Veränderung in der Art, wie Programme organisiert werden – nämlich in Begriffen wie Methoden, Klassen, Instanzen und Vererbung – und lässt sich durch fünf grundlegende Konzepte charakterisieren, die nachfolgend angeführt sind und im Anschluss daran kurz erläutert werden:

- Verbergen von Information durch Datenkapselung in Objekten,
- Objekt-Kommunikation via Nachrichten,
- Polymorphie,
- Klassenkonzept,
- Vererbung von Wissen.

1.1 Datenkapselung in Objekten

Objektorientierte Programmierung ist eine Programmiermethode, bei der Objekte und nicht Funktionen die "agierenden" Software-Komponenten darstellen. Das Objekt bildet die atomare Einheit eines objektorientierten Programmiersystems. Es besitzt eine interne Struktur (deklarativer Part) und verfügt über einen Satz erlaubter Operationen (prozeduraler Part), die als die Methoden des Objekts bezeichnet werden und das Verhalten des Objekts beschreiben. Jedes Objekt hat seinen eigenen "privaten" Speicherbereich, der die objektspezifischen Daten enthält und damit den Zustand des Objekts repräsentiert. Diese "Privatsphäre" ist nur dem Objekt selbst unmittelbar zugänglich, andere Objekte können diesen Privatbereich weder lesen noch verändern, die Implementierung bleibt nach außen verborgen. Das Verstecken von Details wird in diesem Zusammenhang auch Kapselung genannt. Will ein Objekt A ein objektspezifisches Datum eines Objekts B lesen, so ist es darauf angewiesen, dass das Objekt B eine Methode bereitstellt, die diesen Wert liefert.

In [1] steht ergänzend hierzu zum Thema objektorientierte Programmierung (mit CLOS) und Datenkapselung, dass es sich dabei um zwei getrennte Konzepte handelt, obwohl Kapselung oft im Zusammenhang mit objektorientierter Programmierung genannt wird. In COMMON LISP sind Packages eine Möglichkeit, um zwischen privaten und öffentlichen Informationen zu unterscheiden. Um den Zugriff auf irgendetwas einzuschränken, kann

es in ein separates Package gesteckt werden, und nur die Namen werden exportiert, die Teil der externen Schnittstelle sind.

1.2 Objekt-Kommunikation via Nachrichten

Objekte kommunizieren untereinander durch den Austausch von Nachrichten. Ein Objekt wird aktiviert, indem ihm von einem anderen Objekt, das nur die Schnittstelle des zu aktivierenden Objekts nach außen, aber nicht dessen Struktur kennt, eine Nachricht geschickt wird. Stimmt diese Nachricht mit einer der Methoden überein, die dem zu aktivierenden Objekt bekannt sind, wird die zugehörige Methode aufgerufen, die darin spezifizierten Anweisungen werden dann abgearbeitet. In [6] wird eine Methode als Prozedur bezeichnet, die einem Objekt zugeordnet ist und als Reaktion auf eine zugehörige Nachricht ausgeführt wird.

Das Verhalten eines Objekts ist damit beschrieben, wie es auf bestimmte Nachrichten reagiert. Ein Objekt kommuniziert mit seiner Umwelt nur über seine Methoden per Absetzen bzw. Empfangen von Nachrichten.

1.3 Polymorphie

Ein wesentliches Unterscheidungsmerkmal zwischen Nachrichten (= Mitteilungen an Objekte) und Methoden (= Funktionen von Objekten) besteht darin, dass Nachrichten global gültig sind insofern, dass jedem Objekt jede Nachricht geschickt werden darf, die Zuordnung zu der passenden lokalen Methode, so es sie gibt, wird vom empfangenden Objekt vorgenommen. Nachrichten sind programmspezifisch, Methoden objektspezifisch. Die Fähigkeit verschiedener Objekte, auf dieselbe Nachricht unterschiedlich zu reagieren, wird im Kontext der objektorientierten Programmierung als das Phänomen des Polymorphismus bezeichnet.

1.4 Klassenkonzept

Bei der objektorientierten Programmierung erlaubt das Konstrukt der Klasse, Objekte gleicher Struktur und gleichen Verhaltens zu gruppieren. Jedes Objekt gehört einer Klasse an. Das Klassenkonzept erlaubt die einmalige, typisierte Beschreibung der Struktur und des Verhaltens aller gleichgearteten Objekte, die als Instanzen der Klasse erzeugt werden können. Damit braucht nicht jedes Objekt neu beschrieben zu werden, bei der Instantiierung genügt die Angabe der zugehörigen Klasse, wodurch der "Bauplan" festgelegt ist.

1.5 Vererbung von Wissen

Die eigentliche Stärke des Klassenkonzepts liegt in der Möglichkeit zur Hierarchisierung von Klassen und der Vererbung von Wissen entlang der Klassenhierarchie. Vererbung meint im Kontext der objektorientierten Programmierung nichts anderes als die Ableitung einer Klasse aus einer anderen: Die abgeleitete Klasse (= Unterklasse) erbt die Struktur- und Methodenbeschreibung der ihr übergeordneten Klassen (= Oberklassen), sie kann diese geerbten Informationen modifizieren oder erweitern. Inhaltliche Spezialisierungen und Aktualisierungen können so über die Hierarchisierung von Klassen ausgenutzt werden. Bei der Programmierung unterstützt der Vererbungsmechanismus die Nutzung existierender Implementationen.

2 Beispiel

Nachfolgend sei ein kleines COMMON LISP-Programm aus [1] mit Strukturen und einer Funktion zur Berechnung zweidimensionaler Flächen angeführt (Programm 1):

```
(defstruct rechteck
   breite hoehe)

(defstruct kreis
   radius)

(defun flaeche (x)
   (cond ((rechteck-p x)
          (* (rechteck-breite x)
             (rechteck-hoehe  x)))
          (kreis-p    x)
          (* pi
             (expt (kreis-radius x)
                   2))) ))

> (let ((r (make-rechteck)))
       (setf (rechteck-breite r)
             2)
       (setf (rechteck-hoehe  r)
             3)
       (flaeche r))

6
```

Im Vergleich dazu stellt sich die objektorientierte Flächenberechnung mit Klassen und Methoden wie folgt dar (Programm 2):

```
(defclass rechteck ()
   (breite hoehe))

(defclass kreis ()
```

```
   (radius))

(defmethod flaeche ((x rechteck))
   (* (slot-value x
                  'breite)
      (slot-value x
                  'hoehe)))

(defmethod flaeche ((x kreis))
   (* pi
      (expt (slot-value x
                        'radius)
            2)))

> (let ((r (make-instance 'rechteck)))
       (setf (slot-value r
                         'breite)
             2)
       (setf (slot-value r
                         'hoehe)
             3)
       (flaeche r))

6
```

Beim objektorientierten Ansatz wird das Programm in mehrere unterschiedliche Methoden gleichen Namens aufgeteilt, wobei jede für einen bestimmten Argument-Typ bzw. –Klasse zuständig ist. Die beiden Methoden in Programm 2 definieren implizit eine (generische) Funktion FLAECHE, die genauso arbeitet wie die Funktion in Programm 1. Bei Aufruf von FLAECHE in Programm 2 ermittelt der LISP-Interpreter zunächst den Typ bzw. die Klassenzugehörigkeit des ersten Arguments, sucht und ermittelt dann die zuständige Methode und ruft sie schließlich auf.

Zwei wesentliche Stärken des objektorientierten Ansatzes zeigen sich bereits bei diesem kleinen Beispiel:

- Es erfolgt beim Programm 2 eine Komplexitätsreduzierung der Funktion FLAECHE durch Aufsplitten der geforderten Funktionalität in kleinere dedizierte Methoden-Einheiten. Programm 1 stellt dieselbe Funktionalität in *einer* Funktion bereit, die mittels bedingtem Ausdruck für verschiedene Argumente verschiedenen Typs differenzieren muss.
- Die leichte Änderbarkeit und Erweiterbarkeit ist offensichtlich: Zur Erweiterung des Programms 2 für weitere zweidimensionale Flächen können die vorhandenen Klassen- und Methodendefinitionen unverändert bleiben; es bedarf lediglich einer Ergänzung durch weitere zusätzliche Methoden mit dem Namen FLAECHE für entsprechende Argument-Typen bzw. –Klassen. Beim Programm 1 muss hierfür das Coding der vorhandenen Funktion FLAECHE für jeden zusätzlichen Argument-Typ erweitert werden.

3 CLOS-Beschreibung

CLOS besteht aus einer Menge von Tools zur Entwicklung objektorientierter Programme in COMMON LISP.

Die nachfolgenden Abschnitte orientieren sich an dem Aufbau von [2]. Sie geben zunächst eine Einführung in das CLOS-Modell und beschreiben die wesentlichen Bestandteile eines CLOS-Programms. Dann werden - ohne Anspruch auf Vollständigkeit - zentrale Programmiertechniken und CLOS-Features behandelt: Programmierung mit Methoden, Steuerung des 'generic dispatch', Klassenvererbung, Definition und Redefinition von CLOS-Elementen und Erzeugung und Initialisierung von Instanzen.

3.1 Einführung in das CLOS-Modell

Allgemein gilt, dass Programme auf Objekten operieren, die Entitäten der realen Welt oder Abstraktionen repräsentieren. CLOS unterstützt den der Programmierung zugrunde liegenden Modellierungsprozess dadurch, dass es verschiedene Klassen von Objekten auf der Basis deren Struktur und Verhaltens zu beschreiben vermag. Mit CLOS lassen sich Beziehungen zwischen Klassen einfach darstellen, und es bietet Vererbungsmechanismen für Objektstrukturen und -verhalten.

CLOS unterstützt die modulare Entwurfsmethode des Trennens der Implementierung eines Programmmoduls von seiner Schnittstelle nach außen, welche eine übergeordnete operationale Beschreibung bezüglich einer bestimmten Objektmenge bildet. CLOS-Programme/-Module, die andere CLOS-Programme/-Module aufrufen, machen von den über diese Schnittstelle zur Verfügung gestellten Operationen Gebrauch, um Objektdaten zu lesen oder zu ändern, um Objekte zu erzeugen und um andere Methoden auszulösen, ohne dabei die jeweils zugrunde liegende Implementierung zu kennen. Dadurch wird das Design und die Wartung komplexer Programme erleichtert.

3.2 Bestandteile von CLOS-Programmen

Ein CLOS-Programm besteht aus Klassen, (Objekt-)Instanzen, generischen Funktionen und Methoden.

CLOS basiert auf dem Typ-System von COMMON LISP (siehe hierzu etwa [4]). Eine CLOS-Klasse ist eine Datenstruktur, ein definierbarer COMMON LISP-Datentyp, ein CLOS-Klassenname ist ein COMMON LISP-Typspezifikator. .Ein einzelnes Objekt eines solchen Typs wird als Instanz der betreffenden Klasse bezeichnet. Jede Instanz einer vorgegebenen Klasse hat dieselbe (Klassen-)Struktur, zeigt dasselbe (Klassen-)Verhalten und weist denselben (Klassen-)Typ auf, zeichnet sich aber jeweils durch eine eigene Identität und einen eigenen Zustand aus.

Defclass ist der Name des Makros zur Definition einer Klasse. Die Struktur einer Klasse wird mittels Slots, jeder Slot mittels eines Slot-Spezifikators beschrieben. Ein Slot hat einen Namen zur Identifikation und einen Wert, der den aktuellen Zustand eines Slots zu einem bestimmten Zeitpunkt wiedergibt. Diese Zustandsinformation kann mittels spezieller Zugriffsroutinen (engl.: accessors) gelesen und verändert werden. Durch die Definition einer Klasse werden Konstruktor-, Selektor- und Modifikator-Zugriffsroutinen für die Klasse und deren Slots durch CLOS nach Spezifikation automatisch generiert.

Der Rumpf einer Klassendefinition ist nichts anderes als eine Liste von Slot-Spezifikatoren. Ein Slot-Spezifikator kann als ein Symbol gegeben sein, welches den Slot-Namen darstellt, oder als eine Liste, bestehend aus dem Slot-Namen und einer oder mehrerer Slot-Optionen wie z.B. *:accessor*, *:reader*, *:writer*, *:initarg*, *:initform*, *:type* oder *:allocation*.

Es gibt zweierlei Varianten von Slots: lokale Slots, deren Werte jeweils als Teil des Speicherbereichs einer Instanz individuell ausgeprägt sein können, und globale Slots, die jeweils als Teil der Klassenbeschreibung einmalig für alle Instanzen einer Klasse gespeichert sind. Die Slot-Option *:allocation* erlaubt die Auszeichnung eines Slots als lokalen Slot (*:allocation :instance* (= Default)) oder als globalen Slot (*:allocation :class*).

CLOS unterstützt das Prinzip der Vererbung: Eine Klasse erbt die Struktur (die Slots) und das Verhalten (die Methoden) ihrer übergeordneten Klassen und vererbt diese weiter an ihre untergeordneten Klassen.

CLOS-Programme operieren auf Instanzen mittels generischer Funktionen, die sich nach außen wie normale COMMON LISP-Funktionen darstellen und auch wie solche aufzurufen sind. Mit dem Makro *defgeneric* kann eine generische Funktion definiert werden. Eine defgeneric-Form[2] beschreibt lediglich die Schnittstelle einer generischen Funktion nach außen, bestehend aus dem Namen der generischen Funktion, den Parametern[3] und gegebenenfalls einem Dokumentations-String, der textuell wiedergibt, was die Funktion macht und welche Werte sie zurückliefert. Die andere Möglichkeit, eine generische Funktion zu definieren, besteht darin, mittels des Makros *defmethod* eine Methode zu definieren; bei nicht existierender zugehöriger generischer Funktion erzeugt CLOS automatisch eine solche.

Polymorphismus wird durch generische Funktionen möglich. An einem Beispiel - etwa aus dem finanzwirtschaftlichen Bereich des Zahlungsverkehrs - wird dies deutlich: Betrachten wir die Aufgabe bzw. Aktivität der Rückgabe einer Zahlung. Wir können eine generische Funktion RUECKGABE definieren, welche zur Rückgabe beliebiger Zahlungen herangezogen werden kann. Die Funktionsschnittstelle ist für alle Zahlungstypen gleich, die jeweilige Implementierung unterscheidet sich aber in Abhängigkeit beispielsweise vom Zahlungstyp (z.B. Überweisungen, Lastschriften oder

[2] Form: auswertbarer Ausdruck

[3] Ein Parameter ist eine Variable, die während der Ausführung einer Funktion an ein Argument gebunden ist. In anderer Fachliteratur werden Parameter genauer als formale und Argumente als aktuelle Parameter bezeichnet.

Schecks). Bei allen Zahlungstypen müssen Aktivitäten wie Protokollierung, Archivierung oder Stornierung durchgeführt werden; Unterschiede bestehen allerdings z.B. bei der Generierung des Rückgabeverwendungszwecktextes, bei der Gebührenermittlung oder bei durchzuführenden Plausibilitätsprüfungen.

CLOS fordert die Kongruenz der Lambda-Liste[4] einer generischen Funktion mit den Lambda-Listen aller zugehörigen Methoden, d.h. die betreffenden Lambda-Listen müssen dieselbe Anzahl von Parametern aufweisen. Während die Definition einer normalen COMMON LISP-Funktion sowohl die Schnittstelle nach außen als auch die interne Implementierung festlegt, wird durch eine generische Funktion lediglich die Schnittstelle spezifiziert; die Implementierung einer generischen Funktion basiert auf einer Menge von Methoden, die von der Schnittstellendefinition getrennt sind. Welche Methode über die eine Schnittstelle angesprochen wird, hängt von der Klasse der Argumente ab. Eine generische CLOS-Funktion besteht in der Regel aus mehreren Implementierungen, eine Implementierung kann wiederum aus einer oder mehreren Methoden bestehen. Methoden bilden folglich die den generischen Funktionen zugrunde liegenden Implementierungen.

Eine generische Funktion definiert die Schnittstelle einer einzelnen Operation. Alle generischen Funktionen eines Programms bzw. Moduls können zusammen als ein Protokoll bezeichnet werden, welches das vollständige interne und externe Verhalten aller Objekte eines Programms bzw. Moduls spezifiziert.

Die vordefinierte Klasse *standard-object* dient dazu, für das Verhalten von Objekten Default-Methoden bereitzustellen. Für alle benutzer-definierten Klassen (Definition mittels *defclass*) bildet *standard-object* eine übergeordnete Klasse. Die CLOS-Klasse *t* ist die übergeordnete Klasse aller Klassen außer von sich selbst, sie ist die Wurzel des CLOS-Klassensystems und damit auch die einzige Klasse ohne übergeordnete Klasse.

Die Lambda-Liste einer Methode spezifiziert zweierlei Arten von Parametern:

- Ein spezialisierter (formaler) Parameter besteht immer aus einer Liste mit einer Variablen und einem Klassennamen[5] und zeigt die Anwendbarkeit der Methode dadurch an, dass er die Argumentklasse festlegt oder spezialisiert, für die die Methode anwendbar ist. Eine Methode kann einen oder mehrere spezialisierte Parameter besitzen. Liegen mehrere spezialisierte Parameter vor, so wird genauer von einer Multi-Methode gesprochen. Eine (Multi-)Methode ist anwendbar, wenn alle aktuellen Parameter als Argumente des Aufrufs der zugehörigen generischen Funktion den durch die spezialisierten formalen Parameter geforderten Klassen angehören. Als spezialisierte Parameter können beliebige Pflichtparameter ohne Defaultwerte fungieren, nicht aber optionale.
- Ein nicht spezialisierter (formaler) Parameter spezifiziert keine Bedingung für die Methodenanwendbarkeit, sondern stellt lediglich eine Variable dar, die an den betreffenden aktuellen Parameter, d.h. an das korrespondierende Argument

[4] In COMMON LISP ist eine Lambda-Liste der Teil einer Funktion bzw. Methode, der die Namen der Parameter dieser Funktion spezifiziert.

[5] Der Klassenname fungiert als Spezialisierung.

der generischen Funktion, gebunden wird. Ein nicht spezialisierter formaler Pflichtparameter ist äquivalent zu einem spezialisierten Parameter, der das korrespondierende Argument auf die Klasse *t* spezialisiert.

Der Vorgang des Auswählens der aufzurufenden Methode(n) beim Aufruf einer generischen Funktion auf der Basis der Argument-Klasse(n) und das Aufrufen eben dieser Methode(n) wird als 'generic dispatch' bezeichnet. Dieser läuft folgendermaßen ab (was unter Vor-, Haupt- und Nachmethoden zu verstehen ist, ist weiter unten beschrieben):

- Bestimmung der Argumenttypen,
- Lokalisierung aller anwendbarer Methoden,
- Sortierung der anwendbaren Methoden u.a. auf der Basis der Vorrangfolge der zugehörigen Klassendefinition[6],
- Ausführung der Vormethoden in 'most-specific-first'-Reihenfolge,
- Ausführung der - gemäß Vorrangfolgeliste - spezifischsten Hauptmethode,
- Ausführung der Nachmethoden in 'most-specific-last'-Reihenfolge,
- Lieferung des Ergebniswerts der Hauptmethode.

Beim Aufruf einer generischen Funktion erfolgt stets automatisch ein 'generic dispatch', Methoden werden nie direkt, sondern immer nur über die 'generic dispatch'-Prozedur aufgerufen. Was beim Aufruf einer generischen Funktion passiert, hängt von der Klasse der beteiligten Objekte ab (siehe [6]). Der Programmierer verknüpft eine Methode mit der zugehörigen generischen Funktion über die Namensgebung und spezifiziert in der Lambda-Liste der Methode die Klasse bzw. die Klassen, für welche die Methode zuständig ist bzw. sind. Der Aufruf einer generischen Funktion kann die Selektion einer Implementierung zur Folge haben, die aus mehreren Methoden besteht.

In Abhängigkeit des Zeitpunkts des Aufrufs einer Methode im Rahmen des 'generic dispatch' kann zwischen vier verschiedenen Typen oder Rollen von Methoden unterschieden werden: Vormethoden werden, falls spezifiziert, als erste aufgerufen und vollziehen vorbereitende Schritte, die Hauptarbeit macht jeweils die Hauptmethode, Nachmethoden können u.U. notwendige Nacharbeiten vollziehen; schließlich unterstützt CLOS auch Rahmenmethoden, die beispielsweise der Festlegung des (Variablenwerte-)Kontextes für die Ausführung der übrigen Methoden dienen. Die Hauptmethode liefert den Wert der generischen Funktion, die Ausführung der übrigen Methodentypen hat das Erzielen von Seiteneffekten zum Zwecke.

Zur Spezifikation des Methodentyps dienen (Methoden-)Qualifikatoren (z.B. :*after*), welche innerhalb einer Methodendefinition unmittelbar auf den Namen der zugehörigen generischen Funktion folgen.

Zur Lösung von Vererbungskonflikten, die dadurch entstehen können, dass übergeordnete Klassen konkurrierende Charakteristika (Methoden, Slots, Default-Werte o.ä.) spezifizieren, dient die Definition einer Klassenrangfolge. Jede Klassendefinition besitzt eine Rangfolgenliste, die alle direkt übergeordneten Klassen enthält. Die Liste ist nach

[6] Hierzu mehr im nachfolgenden Abschnitt 'Programmierung mit Methoden'.

dem Grad der Spezifität geordnet, der Vererbungsmechanismus orientiert sich vom Spezifischen zum weniger Spezifischen, eine Klasse hat vor deren übergeordneten Klassen Vorrang. Im Gegensatz zu COMMON LISP unterstützt CLOS *multiple* Vererbung, wodurch noch größere Flexibilität erreicht wird.

3.3 Programmierung mit Methoden

Beim Lesezugriff auf einen Slot, dessen Wert dynamisch zur Laufzeit aus anderen Daten zu berechnen ist, sind zweierlei Werteberechnungsmodi möglich: die einmalige Berechnung zum Zeitpunkt der Instantiierung des zugehörigen Objekts, welchem der berechnete Wert vor dem ersten Zugriff auf diese Größe zugewiesen wird, oder die jeweils aktuelle Berechnung zum Zeitpunkt des Datenzugriffs mittels einer Methode bzw. Funktion. Der Vorteil der einmaligen Wertezuweisung zum Instantiierungszeitpunkt besteht in einer schnelleren Datenbereitstellung bei jedem Lesezugriff, die Nachteile sind ein größerer Speicherbedarf für den Objektzustand und eine zeitaufwendigere Objekt-Instantiierung. Die zweitgenannte Vorgehensweise der jeweiligen Werteberechnung zum Lesezugriffszeitpunkt auf den Objekt-Slot zeichnet sich neben der stets garantierten Datenaktualität durch geringeren Speicheraufwand und schnellere Objekt-Instantiierung auf Kosten einer langsameren Datenbereitstellung aus.

Die Eingabe der Slot-Option ':accessor' innerhalb einer Klassendefinition mittels *defclass* veranlasst CLOS, je eine Methode für eine generische Lesefunktion und für eine generische Schreibfunktion zu erzeugen. Dabei handelt es sich um Hauptmethoden. Die generierten Zugriffsfunktionen basieren auf der CLOS-Grundfunktion *slot-value*, die auch für die direkte Programmierung “von Hand” zur Verfügung steht.

Beim Zugriffsversuch auf einen Slot, der noch an keinen Wert gebunden ist, ruft CLOS die Grundfunktion *slot-unbound* zur Ausnahmebehandlung auf. Der Programmierer kann diese generische Funktion, für die eine Default-Methode vorgegeben ist, nach eigenen Bedürfnissen spezialisieren.

Für eine Reihe von COMMON LISP-Typen stellt CLOS gleichnamige 'Built-in'-Klassen zur Verfügung wie beispielsweise für *array*, *symbol* oder *list*. Damit besteht die Möglichkeit, für COMMON LISP-Typen spezialisierte Methoden zu schreiben. 'Built-in'-Klassen sind implementationsabhängig und haben z.B. hinsichtlich Struktur oder Instantiierung etwas eingeschränktere Eigenschaften als solche Klassen, die mit *defclass* definiert werden.

CLOS ermöglicht auch das Spezialisieren von Methoden auf individuelle COMMON LISP-Objekte, nicht nur auf Klassen. Eine Methode, bei der mindestens einer ihrer formalen Pflichtparameter auf ein individuelles COMMON LISP-Objekt spezialisiert ist, wird als individuelle Methode bezeichnet. Die Lambda-Liste einer individuellen Methode enthält mindestens einen spezialisierten Parameter der Form *(var (eql form))*; die Parameterspezialisierung dabei ist der Ausdruck *(eql object)*, wobei *object* das Ergebnis der Evaluierung von *form* ist. Der Ausdruck *form* wird nur einmal zum Zeitpunkt der Makroexpansion der Methodendefinition evaluiert.

Beim Aufruf einer generischen Funktion erfüllt ein aktueller Parameter *arg* die Spezialisierung des korrespondierenden formalen Methoden-Pflichtparameters genau dann, wenn gilt:

(eql T (eql arg 'object)),	sofern der formale Parameter der Gestalt *(var (eql form))* ist, wobei sich der Wert 'object durch Evaluierung des Ausdrucks form zum Zeitpunkt der Makroexpansion der Methodendefinition ergibt,
(eql T (typep arg 'Klassenname)),	sofern der formale Parameter der Gestalt *(var Klassenname)* ist, oder
(eql T (typep arg 't)),	sofern der formale Parameter *var* unspezialisiert ist.[7]

Ein wesentlicher Schritt beim 'generic dispatch' besteht darin, anwendbare Methoden in eine Vorrangreihenfolge zu bringen, wofür CLOS als Entscheidungsgrundlage folgendes wissen muss:

- die Menge der anwendbaren Methoden,
- die Klassenvorrangliste der Klasse eines jeden Pflichtarguments der generischen Funktion und
- die Argument-Vorrangreihenfolge der generischen Funktion.

Zur Vorrangsbestimmung zweier anwendbarer Methoden vergleicht CLOS die Parameterspezialisierungen dieser Methoden, wobei ein unspezialisierter Parameter als ein auf die Klasse t spezialisierter Parameter betrachtet wird. Normalerweise[8] ist die Argument-Vorrangreihenfolge von links nach rechts, was bedeutet, dass CLOS beim Vergleich der Parameterspezialisierungen bei den jeweils ersten Parametern (von links) der zu vergleichenden Lambda-Listen beginnt. Es sind zwei Fälle zu unterscheiden:

Fall 1:
Unterscheiden sich die betrachteten Parameterspezialisierungen, wird der spezifischere Parameter von beiden nach folgender Regel bestimmt:

> Eine Spezialisierung auf ein individuelles Objekt ist spezifischer als eine Spezialisierung auf eine Klasse.
>
> Wenn es sich bei beiden Parameterspezialisierungen um Klassen handelt, basiert die Spezifitätsbestimmung auf den Klassenvorranglisten der korrespondierenden Argumente der generischen Funktion.

Die übrigen Parameterspezialisierungen bleiben außer Betracht.

---

[7] Jedes Argument bzw. jeder aktuelle Parameter beim Aufruf einer generischen Funktion erfüllt einen korrespondierenden unspezialisierten formalen Parameter, da jedes COMMON LISP-Objekt vom Typ *t* ist.

[8] Die Funktionsoption *:argument-precedence-order* der Definition einer generischen Funktion mittels *defgeneric* erlaubt die Festlegung einer anderen Argument-Vorrangreihenfolge. Als Default-Einstellung gilt die Vorrangreihenfolge left-to-right.

Fall 2:
Unterscheiden sich die Parameterspezialisierungen nicht, werden die nächsten beiden Parameterspezialisierungen miteinander verglichen, und es wird gleichermaßen verfahren wie mit den jeweils vorausgegangenen beiden. Zwei Methoden mit gänzlich übereinstimmenden Parameterspezialisierungen müssen unterschiedliche Qualifikatoren haben; in diesem Fall spielt es dann keine Rolle, welche Methode der beiden die spezifischere ist.

3.4 Steuerung des 'generic dispatch'

Der Standardablauf des 'generic dispatch' ist bereits beschrieben worden. CLOS bietet eine Reihe von Möglichkeiten, auf dieses Prozedere Einfluss zu nehmen. Zu unterscheiden sind hierbei deklarative und imperative Techniken. Das Verknüpfen einer Methode mit einer bestimmten Rolle durch einen Qualifikator dient zur deklarativen Steuerung des 'generic dispatch' gleichermaßen wie die Unterstützung sogenannter Methodenkombinationstypen, welche nachfolgend erläutert werden. Imperative Techniken, die eine explizite Steuerung der jeweils als nächste anzuwendende Methode ermöglichen, sind die Definition von Rahmenmethoden und das Aufrufen anwendbarer, aber nicht priorisierter Hauptmethoden. Auch darüber mehr in diesem Abschnitt.

Jede generische Funktion besitzt einen Methodenkombinationstyp, welcher für den 'generic dispatch' das Auswertungsmuster für die anzuwendenden Methoden festlegt. Der Methodenkombinationstyp steuert dreierlei Dinge:

- die zu berücksichtigenden Methodenqualifikatoren und die Rolle, die sie spezifizieren,
- die Reihenfolge, in welcher die anwendbaren Methoden aufgerufen werden, und
- die Art und Weise, in der der Wert der generischen Funktion gebildet wird.

Beim 'generic dispatch' wird aus den anwendbaren Methoden unter Berücksichtigung des zugrunde liegenden Methodenkombinationstyps der betreffenden generischen Funktion COMMON LISP-Code generiert, der die Implementierung der generischen Funktion für die jeweiligen Argumenttypen repräsentiert und als effektive Methode bezeichnet wird. *standard* ist der Default-Methodenkombinationstyp. Er unterstützt Methoden ohne Qualifikatoren (= Hauptmethoden) und solche mit einem der Qualifikatoren *:before*, *:after* und *:around*; der Aufruf von *call-next-method* ist bei diesem Methodenkombinationstyp bei Rahmen- und Hauptmethoden zulässig.

CLOS bietet neben dem Default-Typ *standard* eine Reihe weiterer vorgegebener Methodenkombinationstypen (*+*, *list*, *nconc*, *and*, *max*, *or*, *append*, *min* und *progn*), die allerdings allesamt keine Vor- und Nachmethoden erkennen. Die *defgeneric*-Option *:method-combination* erlaubt die Angabe eines Methodenkombinationstyps. In der Definition der zu kombinierenden Methoden bedarf es darüber hinaus der Angabe des

Methodenkombinationstyps im Anschluss an den Methodennamen; auch hierbei wird von einem Methodenqualifikator gesprochen Bei der Betrachtung der vorgegebenen CLOS-Methodenkombinationstypen fällt auf, dass sie dieselben Namen wie COMMON LISP-Funktionen bzw. -Operatoren haben. Man spricht daher auch von Operator-Methodenkombinationstypen. Ihre jeweilige Semantik bezüglich der Wertermittlung der generischen Funktion wird durch die gleichnamige COMMON LISP-Funktion definiert, die Hauptmethoden werden in 'most-specific-first'-Reihenfolge kombiniert. Während bei der *standard*-Methodenkombination unqualifizierte Methoden als Hauptmethoden interpretiert werden, sind unqualifizierte Methoden bei der Operator-Methodenkombination nicht zulässig, die Angabe des Operators als *:method-combination*-Option ist notwendig. Ein weiterer Unterschied zur *standard*-Methodenkombination besteht darin, dass der Aufruf von *call-next-method* in Hauptmethoden zu einer Fehlermeldung führt. Bei Rahmenmethoden (s.u.), die bei der Operator-Methodenkombination als solche erkannt und berücksichtigt werden, ist der Aufruf von *call-next-method* zulässig. Mit dem CLOS-Makro *define-method-combination* können überdies neue Operator-Methodenkombinationstypen definiert werden, wenn die vorgegebenen Typen nicht ausreichend sind.

Mit dem Qualifikator *:around* wird eine Methode als Rahmenmethode spezifiziert. Genau genommen stellt die Definition einer Rahmenmethode eine hybride Steuerungstechnik für den 'generic dispatch' dar: Sie ist deklarativ insofern, als der *:around*-Qualifikator eine Methodenrolle spezifiziert (s.o.), sowie auch imperativ, da im Methodenrumpf spezifizierbar ist, welche Methoden als nächste im Rahmen des 'generic dispatch' aufgerufen werden. Sind beim Aufruf einer generischen Funktion zugehörige Rahmenmethoden anwendbar, gestaltet sich der 'generic dispatch' so, dass CLOS die spezifischste Rahmenmethode als erste aller anwendbaren Methoden aufruft. Der evaluierte Wert ist im Gegensatz zu Vor- oder Nachmethoden gleichzeitig der Wert des Aufrufs der generischen Funktion; die übrigen anwendbaren Methoden werden in Rahmenmethode eingebettet. Ruft eine Rahmenmethode die CLOS-Funktion *call-next-method* nicht auf, werden keine weiteren Methoden mehr aufgerufen. Andernfalls wird die nächst weniger spezifische Rahmenmethode aufgerufen, sofern mindestens eine weitere anwendbare Rahmenmethode existiert; gibt es keine weiteren passenden Rahmenmethoden, geht CLOS zum oben beschriebenen Standardablauf des 'generic dispatch' über, d.h. Aufruf anwendbarer Vor-, Haupt- und Nachmethoden.

Eine anwendbare, aber aufgrund geringerer Spezifität nicht priorisierte Hauptmethode, die "im Schatten" einer spezifischeren ist, kann durch *call-next-method* entgegen dem Standard-'generic dispatch' innerhalb der spezifischeren Methode aufgerufen werden. Damit steht ein weiteres, rein imperatives Instrument zur Steuerung des 'generic dispatch' zur Verfügung. Die CLOS-Prädikatsfunktion *next-method-p* erlaubt das Prüfen auf Vorhandensein einer weiteren anwendbaren Hauptmethode.

Bei der Verwendung deklarativer Steuerungstechniken für den 'generic dispatch' ist die Auswertungsreihenfolge anwendbarer Methoden vorhersehbar, ohne Kenntnisse der einzelnen Methodenimplementierungen haben zu müssen. Sie stützt sich ausschließlich auf die Methoden-Qualifikatoren (*:before, :after, :around, :method-combination x*), die die Eignung bzw. die Rolle der involvierten Methoden spezifizieren. Die imperative

Technik ist methoden-implementierungsspezifisch, sie erlaubt den einzelnen Methoden selbst, die beim 'generic dispatch' einzuschlagende Richtung durch den Aufruf von *call-next-method* zu bestimmen. Die damit erzielbare Flexibilität geht auf Kosten der gerade bei der objektorientierten Programmierung angestrebten Modularität. Daher sollte die Verwendung von Rahmenmethoden und *call-next-method* bei der Programmierung in CLOS auf das Notwendigste beschränkt bleiben.

3.5 Klassenvererbung

CLOS kennt zwei Default-Klassen, die vor allem zum Zwecke der Vererbung wichtig sind: *t* und *standard-object*.

Die Klasse *t* ist die übergeordnete Klasse aller benutzerdefinierten und aller in CLOS vorgegebenen Klassen mit Ausnahme von *t* selbst. Damit ist CLOS in der Lage, Default-Charakteristika der Klasse *t* an alle Klassen zu vererben. Ein Effekt der Vererbung von *t* besteht darin, dass jedes COMMON LISP-Objekt auch vom Typ *t* ist. *t* ist Datentyp und Klasse gleichermaßen: Der Datentyp *t* ist die Wurzel des COMMON LISP-Typsystems, die Klasse *t* ist die Wurzel des CLOS-Klassensystems. Alle Klassen haben *t* als unspezifischste und damit als letzte angeführte Klasse in ihrer Klassenvorrangliste. Die Klasse *t* besitzt keine Slots.

standard-object ist auch eine übergeordnete Klasse aller benutzerdefinierten, nicht aber von vorgegebenen Klassen. Damit ist CLOS in der Lage, Default-Verhalten der Klasse *standard-object* speziell an alle benutzerdefinierten Klassen zu vererben. Alle benutzerdefinierten Klassen haben *standard-object* als zweitletzte angeführte Klasse in ihrer Klassenvorrangliste. Auch die Klasse *standard-object* besitzt wie *t* keine Slots.

CLOS ermittelt die Klassenvorrangliste für jede Klasse auf der Basis ihrer Definition und der Definition ihrer übergeordneten Klassen. Die Klassenvorrangliste einer Klasse enthält jeweils die Klasse selbst und alle ihre übergeordneten Klassen. Entscheidend ist die Reihenfolge der Klassen in den Listen der direkt übergeordneten Klassen innerhalb der Definitionen der involvierten Klassen: Die Spezifität ist dabei abnehmend. Eine Klasse hat immer Vorrang vor ihren übergeordneten Klassen. Damit kann eine Klasse Struktur- und Verhaltensaspekte ihrer übergeordneten Klassen übernehmen, überschreiben oder ändern bzw. erweitern. Die spezifischste Klasse innerhalb der Vorrangliste einer Klasse ist immer eben diese Klasse selbst., die Klasse *t* ist die unspezifischste Klasse einer jeden Klassenvorrangliste. Für benutzerdefinierte Klassen gilt, dass die Klasse *standard-object* nach der Klasse *t* die unspezifischste ist.

Die Klassenvererbung legt bei der Konzeption als Basis für die Implementierung eines CLOS-Programms einen Top-Down-Entwurf nahe: Ausgehend von einer oder mehreren Basisklassen werden spezialisierte Klassen abgeleitet. Erwünschte Charakteristika, also Struktur und Verhalten, werden übernommen bzw. geerbt, unerwünschte überschrieben. Erbbar sind Methoden, Slots und Slot-Optionen, wobei letztere differenzierten Vererbungsregeln unterliegen.

3.6 Definition und Redefinition von CLOS-Elementen

Klassen können in beliebiger Reihenfolge definiert werden. Eine Klasse kann vor deren übergeordneten Klassen definiert werden. Ebenso können Methoden und generische Funktionen in beliebiger Reihenfolge definiert werden. Wird eine Methode vor der zugehörigen generischen Funktion definiert, erzeugt CLOS automatisch die generische Funktion. Es gibt in CLOS auch zu berücksichtigende Reihenfolgeabhängigkeiten. Eine Klasse und alle ihre übergeordneten Klassen müssen vor der Instantiierung eben dieser Klasse definiert sein. Wird eine Methode für eine bestimmte Klasse definiert, so muss die Definition dieser Klasse vorausgegangen sein.

Durch die (automatische) Definition einer generischen Funktion, d.h. durch die Auswertung einer *defmethod*- oder einer *defgeneric*-Form wird über die Lambda-Liste das Parameter-Muster für alle zugehörigen Methoden festgelegt: Es spezifiziert die Anzahl der obligatorischen und der optionalen Parameter und die Verwendung von *&rest*- und *&key*-Parametern. Durch eine *defgeneric*-Form wird die minimale Menge von Schlüsselwort-Parametern beschrieben; jede zugehörige Methode kann darüber hinaus weitere Schlüsselwort-Parameter angeben. Im Gegensatz zu generischen Funktionen können Methoden Default-Werte für optionale Parameter vorgeben.

Die interne Repräsentation der CLOS-Objekte *Klasse*, *generische Funktion* und *Methode* sind entsprechende COMMON LISP-Objekte. Die CLOS-Programmierschnittstelle untergliedert sich in zwei Ebenen: Die Makro-Ebene (*defclass*, *defgeneric*, *defmethod*) erlaubt die Definiton und Benennung von CLOS-Objekten; implementiert wird diese Ebene auf der Basis der funktionalen Ebene, die mit den Objekten direkt, nicht über Namen hantiert. Während die Makro-Ebene komfortablere Zugriffe mit einfacherer Syntax ermöglicht, bietet die funktionale Ebene mehr Flexibilität wie beispielsweise die Unterstützung anonymer Klassen und generischer Funktionen.

Bei der Redefinition einer Klasse werden automatisch durch CLOS alle bereits bestehenden Instanzen dieser Klasse gemäß der neuen Definition aktualisiert. Existiert bereits zum Zeitpunkt der Evaluierung einer *defclass*-Form eine Klasse desselben Namens, so ersetzt die neue Klassendefinition die alte. Das Ändern der Definition einer Klasse wirkt sich auch auf alle untergeordneten Klassen sowie deren Instanzen aus, d.h. die Änderungen werden vererbt. Auch die Definition von Zugriffsmethoden aktualisiert CLOS, falls dies erforderlich wird.

CLOS bietet die Möglichkeit, Aktionen zu spezifizieren, die mit der Änderung von Instanzen einhergehen müssen. Ein Beispiel hierfür ist die Wertefestlegung eines Slots mit geändertem Namen mit dem Wert des Slots vor der Namensänderung.

Die Evaluierung einer *defmethod*-Form ersetzt - sofern existent - eine gleichnamige Methode derselben generischen Funktion mit denselben Parametern. Entsprechendes gilt für die Evaluierung einer *defgeneric*-Form; sind dabei allerdings bereits Methoden

definiert, deren Parameter zur Lambda-Liste der neuen generischen Funktion nicht kongruent sind, erzeugt CLOS eine Fehlermeldung.

Mit *defgeneric* lässt sich keine gewöhnliche COMMON LISP-Funktion, kein Makro und keine spezielle Form redefinieren. Hingegen ist es möglich, mittels *defun* eine generische Funktion zu redefinieren.

Durch Aufruf von *change-class* kann die Klassenzugehörigkeit einer bereits existierenden Instanz geändert werden.

3.7 Erzeugung und Initialisierung von Instanzen

Die generische CLOS-Funktion *make-instance* dient zum Erzeugen von Klassen-Instanzen. Die Argumente beim Aufruf von *make-instance* sind zum einen der Name oder das Objekt der Klasse, von der eine Instanz gebildet werden soll, und optional weitere Initialisierungsargumente.

Wenn *make-instance* aufgerufen wird, vollzieht CLOS mehrere Schritte. Zunächst ergänzt es die gegebenenfalls als Initialisierungsargumente übergebenen Werte mit den Initialisierungs-Default-Werten der Klassendefinition. Dann prüft es die Richtigkeit der so erzeugten Liste der Initialisierungsargumente und gibt im Fehlerfall eine entsprechende Meldung aus. Bei korrekter Initialisierungsargumente-Liste weist CLOS den erforderlichen Speicher zu und erzeugt daraufhin eine Instanz mit noch ungebundenen Slots. Die Slot-Bindung erfolgt durch Anwendung der generischen Funktion *initialize-instance* auf die neu generierte Instanz mit der Initialisierungsargumente-Liste. Als Wert der Anwendung von *make-instance* gibt CLOS die initialisierte Instanz zurück.

Die CLOS-Default-Methode für *initialize-instance* sorgt für die Slot-Bindung wie folgt:

Zunächst werden die Initialisierungsargument-Optionen, falls vorhanden, berücksichtigt. Dabei haben die explizit beim Aufruf von *make-instance* als Argumente, korrespondierend zu den *:initarg*-Slot-Optionen, übergebenen Vorrang vor den *:default-initargs*-Klassen-Optionen. An dritter Stelle kommen, falls keine Initialisierungsargumente herangezogen werden können, die *:initform*-Slot-Optionen. Während die *:default-initargs*-Option einen Default-Wert für ein Initialisierungsargument darstellt, bildet die *:initform*-Option einen Default-Wert für einen Slot. Gibt es für einen Slot einer gerade in Entstehung befindlichen Instanz weder ein (Default-)Initialisierungsargument noch einen *:initform*-Default-Wert, so bleibt dieser Slot auch nach Anwendung von *initialize-instance* ungebunden.

Es ist möglich, aber nicht ratsam, die Default-Methode für *initialize-instance* durch eine individuelle Fassung zu ersetzen. Sinnvoll ist unter Umständen die Defintion weiterer Nachmethoden, die sich unmittelbar der Auswertung von *initialize-instance*, gesteuert durch den 'generic dispatch', anschließen.

4 Literatur

[1] Graham, P.: The ANSI COMMON LISP Book. ISBN 0-133-70875-6, US Imports & PHIPEs, 1995.

[2] Keene, S. E.: Object-Oriented Programming in COMMON LISP. A Programmer's Guide to CLOS. ISBN 0-201-17589-4, Addison-Wesley, 1989.

[3] Kienzler, F.: Synthese versus Analyse in modellbasierten KI-Planungssystemen? DIAKON – ein auto-adaptiver diagnostischer Lösungsansatz für Aktionsplanungs- und Konfigurierungsprobleme, Dissertation, Universität Ulm. ISBN 3-8265-6896-6, Shaker-Verlag, 2000.

[4] Mayer, O.: Programmieren in COMMON LISP. ISBN 3-8602-5710-2, Spektrum Akad. Verlag, 1995.

[5] Steele Jr., G. L.: COMMON LISP – The Language. Second Edition. ISBN 1-55558-041-6, Digital Press, 1990.

[6] Winston, P. H., Horn, B. K. P.: LISP. Third Edition. ISBN 0-20108-319-1, Addison-Wesley, 1988.