GMD Research Series

GMD –
Forschungszentrum
Informationstechnik
GmbH

Harry Bretthauer

Entwurf und Implementierung effizienter Objektsysteme für funktionale und imperative Programmiersprachen am Beispiel von Lisp

N° 4/1999

GMD – Forschungszentrum Informationstechnik GmbH
Schloß Birlinghoven
D-53754 Sankt Augustin, Germany
Telefon +49 -2241 -14 -0
Telefax +49 -2241 -14 -2618
http://www.gmd.de

Die vorliegende Veröffentlichung entstand im/
The present publication was prepared within:
Institut für Autonome intelligente Systeme (AiS)
Institute for Autonomous intelligent Systems
http://ais.gmd.de/

Anschrift des Verfassers/Address of the author:
Harry Bretthauer
Deutsche Telekom AG
Technologiezentrum Darmstadt
Am Kavalleriesand 3
D-64295 Darmstadt
E-mail: bretthauer@tzd.telekom.de

Die Deutsche Bibliothek - CIP-Einheitsaufnahme:
Bretthauer, Harry:
Entwurf und Implementierung effizienter Objektsysteme für funktionale und imperative Programmiersprachen am Beispiel von Lisp /
Harry Bretthauer. GMD – Forschungszentrum Informationstechnik GmbH. - Sankt Augustin : GMD – Forschungszentrum Informationstechnik, 1999
(GMD Research Series ; 1999, No. 4)
Zugl.: Saarbrücken, Univ., Diss., 1999
ISBN 3-88457-353-5

ISSN 1435-2699
ISBN 3-88457-353-5

Abstract

Up to now a gap is evident in *object systems* of *functional and procedural programming languages*. The most expressive object system developed in the family of functional languages is CLOS with its outstanding *metaobject protocol*. Its *performance*, however, does not meet the users' needs. In the family of procedural languages the most efficient object system developed is C++. But its support of central *concepts of object-oriented programming*, such as *specialization* and *generalization* of object classes, is not sufficient. This also applies in some degree for JAVA.

Using Lisp as an example this thesis shows how *efficient object systems* can be *designed* and *implemented* so that simple constructs have no overhead because of the presence of complex concepts such as the *metaobject protocol* or the *redefinition of classes*. In contrast to former assumptions, this thesis proofs for the first time that the above mentioned concepts can be realized *without embedding an interpreter* or an incremental compiler in the run-time environment. Therefore, they can also be supported in traditional compiler-oriented programming languages such as Ada, Pascal, Eiffel, C++, and JAVA.

Keywords: object systems, concepts of object-oriented programming, specialization, generalization, metaobject protocol, redefinition of classes, functional and procedural programming languages, performance, interpreter, incremental compiler.

Kurzfassung

Bisherige *Objektsysteme funktionaler und imperativer Programmiersprachen* weisen eine Lücke auf. Aus der funktionalen Tradition wurde das ausdrucksstärkste Objektsystem CLOS entwickelt, das insbesondere durch sein *Metaobjektprotokoll* hervorsticht, dessen *Performanz* aber zu wünschen übrig läßt. Auf der anderen Seite zeichnet sich C++ als besonders effizient aus, unterstützt aber zentrale *Konzepte objektorientierter Programmierung* wie *Spezialisieren* und *Generalisieren* von Objektklassen nur unzureichend, was abgeschwächt auch für JAVA gilt.

In dieser Arbeit wird am Beispiel von Lisp gezeigt, wie man *effiziente Objektsysteme* unter Berücksichtigung des *Verursacherprinzips* so *entwirft* und *implementiert*, daß einfache Konstrukte keinen Overhead durch die Präsenz aufwendiger Konzepte, wie des *Metaobjektprotokolls* oder des *Redefinierens von Klassen*, mittragen müssen. Entgegen bisherigen Annahmen wird hier erstmals nachgewiesen, daß diese Konzepte auch *ohne Quellcodeinterpretation* bzw. -kompilation zur Laufzeit realisiert und somit auch in traditionellen, compiler-orientierten Programmiersprachen, wie Ada, Pascal, Eiffel, C++ und natürlich JAVA, unterstützt werden können.

Schlagwörter: Objektsystem, Konzepte objektorientierter Programmierung, Spezialisieren, Generalisieren, Metaobjektprotokoll, Redefinieren von Klassen, funktionale und imperative Programmiersprachen, Performanz, Quellcodeinterpretation, Kompilation zur Laufzeit.

Zusammenfassung

Objektorientierte Sprachkonstrukte kann man inzwischen in fast allen Programmiersprachen finden. Bei genauerer Betrachtung stellt man jedoch fest, daß es weder einen Konsens über die zu unterstützenden Konzepte noch eine einheitliche Terminologie gibt.

Im Spektrum objektorientierter sowie funktionaler und imperativer Programmiersprachen mit entsprechenden objektorientierten Erweiterungen lassen sich zwei Entwicklungsstränge identifizieren. Der eine Strang ist vom Bemühen um problemadäquate Ausdrucksmittel geprägt, oft unter Vernachlässigung des Performanzaspekts. Hierzu zählen Sprachen wie Smalltalk und CLOS. Der andere Strang stellt Effizienzfragen in den Vordergrund und berücksichtigt Erkenntnisse und Anforderungen aus der objektorientierten Wissensrepräsentation nur unzureichend. Hier ist vor allem C++, aber auch JAVA zu nennen. Es stellt sich daher die *Frage*, ob die Lücke zwischen den beiden Strängen geschlossen und der Gegensatz zwischen Erfordernissen einer problemadäquaten und somit anwendungsnahen objektorientierten Sprache und den Restriktionen einer für die gegenwärtige Rechnergeneration effizient implementierbaren Sprache überwunden werden kann.

Das **Ziel** dieser Arbeit ist also der *Entwurf und die Implementierung von Objektsystemen für funktionale und imperative Programmiersprachen*, die den Anforderungen komplexer Anwendungen bezüglich konzeptueller Klarheit, Flexibilität und Effizienz standhalten. Am Beispiel von LISP mit seinen bekanntesten Objektsystemen FLAVORS und CLOS wird hier schrittweise gezeigt, wie man die Effizienz dieser Objektsystemen verbessern kann, ohne ihre wesentlichen objektorientierten Konzepte aufzugeben und auf das Niveau von C++ zurückzufallen. Allerdings muß signifikanten Effizienzverbesserungen eine kritische Reflektion dieser Konzepte vorausgehen.

Ausgehend von den Defiziten existierender Objektsysteme sowie den Anforderungen komplexer, insbesondere wissensbasierter, Systeme habe ich **12 Kriterien** formuliert, an denen alle Entwurfsentscheidungen zu messen sind: Abstraktion, Klassifikation, Spezialisierung, Komposition, Modularisierung, Orthogonalität, Einfachheit, Reflektion, Erweiterbarkeit, Robustheit, Effizienz und das Verursacherprinzip.

Diese Kriterien müssen auf unterschiedliche Weise berücksichtigt werden. Die Unterstützung der Klassifikation, der Spezialisierung, der Reflektion und der Erweiterbarkeit ist die ureigenste Aufgabe eines Objektsystems. Komposition und Modularisierung stellen dazu orthogonale Prinzipien dar. Ein Modulkonzept sollte zusätzlich zum Objektsystem vorhanden sein und es sollte benutzt werden, um das Objektsystem zu modularisieren. Orthogonalität und Einfachheit beziehen sich auf die Gestaltung der Grundkonzepte und entsprechender Sprachkonstrukte. Ein Objektsystem muß robust gegenüber seinen Erweiterungen sein. Reflektions- und Spezialisierungsmittel dürfen die Integrität des Ob-

jektsystems und seiner Anwendungen nicht gefährden. Die Effizienz eines Objektsystems bezieht sich vor allem auf die statische und dynamische Objekterzeugung, auf Feldzugriffe und den generischen Dispatch einschließlich der Methodenkombination. Um ein effizientes Verhalten komplexer Systeme zu erreichen, brauchen Programmierer ein transparentes Effizienzmodell der Programmiersprache. Die Sprachkonstrukte des Objektsystems müssen daher das Verursacherprinzip einhalten und dieses dem Programmierer bewußt machen. Schließlich muß das Sprachdesign verschiedene Implementierungstechniken unterstützen, um eine umfassende Effizienz zu ermöglichen.

Auf der Grundlage dieser Kriterien und der bisherigen Objektsysteme habe ich objektorientierte *Sprachkonzepte* für ΤΕΛΟΣ entwickelt, die zum einen bessere Ausdrucksmöglichkeiten schaffen und zum anderen effizienter implementierbar sind, z. B. im Vergleich zu CLOS.

Objektorientierte Sprachkonzepte lassen sich unter vier Aspekten diskutieren: der *Objektstruktur*, dem *Objektverhalten*, der *Vererbung* und der *Reflektion*. Die ersten drei Aspekte können in einer Ebene behandelt werden, der sogenannten *Objektebene*. Die Objektstruktur und das Objektverhalten sind der Gegenstand von Vererbungskonzepten. Mit der Reflektion wird eine weitere Ebene betreten, die sogenannte *Metaobjektebene*. Zusammen erhält man eine metareflektive Systemarchitektur. Auf der Metaobjektebene werden Metaobjekte zu Objekten erster Ordnung. Wie man auf der Objektebene seine Anwendungsklassen erweitert und spezialisiert, so kann man auf der Metaobjektebene das Objektsystem der Programmiersprache erweitern und spezialisieren. Diese neue Dimension objektorientierter Programmierung wurde bisher nur in CLOS richtig unterstützt. Irrtümlicherweise glaubte man bisher, als Voraussetzung dafür die Quellcodeinterpretation bzw. -kompilation zur Laufzeit zu benötigen. Mit dieser Arbeit habe ich erstmals nachgewiesen, daß dies nicht der Fall ist. Damit sind die Grundlagen geschaffen worden, um die Konzepte der Spracherweiterungsprotokolle auch in traditionellen, compiler-orientierten Programmiersprachen, wie Ada, Pascal, Eiffel, C++ und natürlich JAVA, zu unterstützen.

Die *Klassifikation der Struktur und des Verhaltens* von Objekten steht im Mittelpunkt objektorientierter Programmierung. Mit der Unterstützung des *Generalisierens und Spezialisierens* von Objektklassen wird die Hauptaufgabe eines Objektsystems angesprochen:

- Um die Objektstruktur spezialisieren zu können, müssen die Felder der Superklassen ererbt und ihre Annotationen wie z. B. der Default-Wert spezialisiert werden können. Dies ist weder in Smalltalk noch in C++ noch in JAVA möglich. In diesen Sprachen können nur neue Felder definiert werden, ererbte Felder werden unspezialisiert übernommen.

- Um das Objektverhalten spezialisieren zu können, müssen Methoden der Superklassen ererbt und speziellere Methoden definiert werden können. Dabei muß es möglich sein, die ererbte Methode auf abstrakte Weise wiederzuverwenden und zu ergänzen. Dazu braucht man mindestens die einfache (explizite) Methodenkombination mit `callNextMethod` als die Generalisierung von `super()` aus Smalltalk sowie `inner` aus BETA. Dies ist in C++ nicht gegeben. Im Unterschied zu CLOS unterscheide ich in ΤΕΛΟΣ zwischen *expliziter und impliziter Methodenkombination*, um z. B. JAVA-Konstruktoren nicht als Einzelfall, sondern als Methoden mit impliziter Methodenkombination zu subsumieren.

Die *Vererbung* regelt, wie das Generalisieren und Spezialisieren von statten geht. Als ein wesentliches Resultat dieser Arbeit wurde die Mixin-Vererbung entwickelt, um die Nachteile der einfachen und (allgemeinen) multiplen Vererbung zu überwinden. Dabei wird zwischen wesentlichen und zusätzlichen Aspekten von Objekten unterschieden. Letztere können mehrfach ererbt werden, in Analogie zur Aneinanderreihung mehrerer Adjektive in der natürlichen Sprache:

- *Basisklassen*, die wesentliche Aspekte repräsentieren, bilden untereinander eine einfache Vererbungshierarchie.
- *Mixin-Klassen*, die zusätzliche Aspekte repräsentieren, können mehrfach geerbt werden. Allerdings kann *eine* Basisklasse nur *disjunkte* Mixin-Klassen erben.

Effiziente Implementierungen von Objektsystemen setzen ein entsprechendes Sprachdesign voraus. Dieses sollte idealerweise fünf Implementierungstechniken unterstützen:

- Modul-Kompilation,
- Applikations-Kompilation,
- inkrementelle Kompilation,
- Bytecode-Kompilation bzw. -Interpretation und
- Einbettungstechnik.

Dabei kann die Bytecode-Kompilation auch unter der Modul-Kompilation subsumiert werden. Außerdem profitiert die Einbettungstechnik von allen Formen der Kompilation der Basissprache. Daß auch Sprachen der Lisp-Familie von der Modul- und Applikations-Kompilation profitieren können, wurde im Projekt APPLY nachgewiesen. Der dafür notwendige Verzicht auf Quellcodeinterpretation und -kompilation zur Laufzeit reicht im Prinzip aus, um traditionelle Kompilationstechniken erfolgreich anzuwenden. Im Verzicht auf Quellcodeinterpretation liegt auch der Schlüssel für die Effizienz meiner Implementierung.

Im Unterschied zu CLOS, daß mit seinen Redefinitionsmöglichkeiten primär auf inkrementelle Kompilation ausgerichtet ist, unterstützt ΤΕΛΟΣ alle genannten Implementierungstechniken. Insbesondere trägt die strikte Unterscheidung zwischen Sprachkonstrukten der Objekt- und der Metaobjektebene sowie die Modularisierung des Metaobjektprotokolls dazu bei, auch die Modul- und Applikations-Kompilation zu unterstützen, was ein Novum für metareflektive Objektsysteme darstellt.

Aber selbst mit der Einbettungstechnik und ohne explizite Verwendung des inkrementellen Compilers ist meine ΤΕΛΟΣ-Implementierung insgesamt effizienter als die derzeit besten kommerziellen CLOS-Realisierungen. Dies ist das Resultat eines konsequenten Sprachdesigns und einer einfachen Implementierungstechnik unter Beachtung des Verursacherprinzips. Dies hat insbesondere am Beispiel des *Redefinitionskonzepts* in ΤΕΛΟΣ dazu beigetragen, den entsprechenden Overhead auf Objekte zu beschränken, die diese Funktionalität auch wirklich benötigen. Darüberhinaus konnte die *Objekterzeugung* und die *Methodenkombination* in ΤΕΛΟΣ wesentlich effizienter realisiert werden als in CLOS.

Danksagung

Diese Arbeit entstand nicht über nacht, es waren eher viele Nächte. Voraus- und mitgegangen war die Hilfe vieler Menschen, denen ich besonders danken möchte.

Thomas Christaller weckte meine Begeisterung für Lisp, initiierte das Thema und betreute die Arbeit bei ihrer Durchführung. Wolfgang Wahlster half durch viele kritische Hinweise in der Abschlußphase, die Kernaussagen auf den Punkt zu bringen. Beiden danke ich für die Betreuung und Begutachtung dieser Arbeit.

Mein besonderer Dank gilt Jürgen Kopp: für die erfolgreiche Zusammenarbeit in gemeinsamen Projekten in der GMD, für den stetigen Ansporn beim Verfassen der Arbeit und für die vielen kleinen und großen Tips, die mir viel Zeit erspart haben. Jörg Johnson-Schaaf danke ich für die interessanten Gespräche bei der Zusammenarbeit im KIKon-Projekt, die immer wieder auch grundlegende Fragen dieser Arbeit berührten.

Jürgen Walther und Eckehard Groß danke ich für ihre Bereitschaft, immer wieder neue Versionen von MCS zu testen, und für die hilfreichen Anregungen aus den Erfahrungen mit der KI-Werkbank BABYLON.

Harley Davis, Jürgen Kopp, Greg Nuyens, Julian Padget und Keith Playford möchte ich für die intensiven und fruchtbaren Diskussionen bei der Definition von Τελος als integralem Bestandteil von EuLisp danken.

Wolfgang Goerigk, Ulrich Hoffmann, Heinz Knutzen, Ingo Mohr und Friedemann Simon lieferten im Rahmen des APPLY-Projekts interessante Diskussionsbeiträge zur inkrementellen, Modul- und Applikationskompilation von Lisp.

Wolfgang Goerigk danke ich für die kritischen Fragen zum roten Faden der Arbeit und Ulrich Hoffmann für die Unterstützung bei den abschließenden Performanztests.

Eva-Maria Bretthauer, Wolfgang Gräther, Joachim Hertzberg, Jürgen Kopp und Ulrike Petersen gaben wichtige Anmerkungen zum Stil und Inhalt der Arbeit.

Dankend erwähnen möchte ich auch die Unterstützung durch Werner Gries, Bernd Reuse und Klaus Rupf bei der Wiederaufnahme meiner wissenschaftlich-technischen Arbeit nach drei Jahren im BMBF.

Abschließend danke ich allen Kollegen im Forschungsbereich Künstliche Intelligenz der GMD sowie bei der Dialogis Software & Services GmbH, die in der einen oder anderen Form zum Gelingen dieser Arbeit beigetragen haben.

Inhaltsverzeichnis

1 Einleitung **1**

1.1 Motivation 1

1.2 Ziel und Arbeitshypothese 4

1.3 Hintergrund und Aufbau der Arbeit 5

2 Wie entwirft man effiziente Objektsysteme? **7**

2.1 Beispiele komplexer Softwaresysteme: BABYLON, TINA, KIKon 9

2.1.1 KI-Werkbank BABYLON 9

2.1.2 TINA 12

2.1.3 KIKon 12

2.2 Anforderungen aus dem Softwarelebenszyklus komplexer Anwendungen . . 13

2.2.1 Komplexität von Softwaresystemen 13

2.2.2 Anforderungen aus Analyse und Design 14

2.2.3 Anforderungen aus Implementierung, Evolution und Wartung 15

2.3 Designkriterien für objektorientierte Programmiersprachen 15

2.3.1 Abstraktion 16

2.3.2 Klassifikation 17

2.3.3 Spezialisierung und Generalisierung 17

2.3.4 Komposition 18

2.3.5 Hierarchiebildung 18

2.3.6 Modularisierung 18

2.3.7 Reflektion 19

2.3.8 Erweiterbarkeit und Wiederverwendbarkeit 19

2.3.9 Einfachheit 20

2.3.10 Orthogonalität 21

2.3.11 Robustheit 21

2.3.12 Effizienz 22

2.3.13 Verursacherprinzip 23

2.3.14 Zusammenfassung der Kriterien 23

3 Stand der Technik **25**

3.1 Programmiersprachen, Verarbeitungsmodelle und Programmierstil 25

3.1.1 Abstraktion in Programmiersprachen 26

3.1.2 Verarbeitungsmodelle und Programmierstil 27

3.2 Objektorientierung: Ein Sammelsurium von Begriffen und Konzepten 30

3.2.1 Wiederverwendbarkeit und Wartbarkeit als Motivation 31

3.2.2 Objekte, Klassen und Instanzen 31

3.2.3 Kapselung, Information-Hiding und Modularisierung 32

3.2.4 Vererbungskonzepte 33

3.2.5 Delegieren 35

3.2.6 Senden von Nachrichten 35

3.2.7 Überladen von Operatoren und Polymorphie 36

3.2.8 Methodenkombination 37

3.2.9 Reflektion und Metaobjektprotokolle 38

3.3 Ausgewählte verwendete Objektsysteme 39

3.3.1 Smalltalk 40

3.3.2 C++ 42

3.3.3 Java 45

3.4 Objektsysteme für Lisp 47

3.4.1 MFS: Micro Flavor System 49

3.4.2 ObjVLisp 51

3.4.3 CLOS: Common Lisp Object System 53

3.5 Zusammenfassung 66

4 Objektorientierte Sprachkonzepte **71**

4.1 Orthogonale Sprachaspekte 72

4.1.1 Verteilte Systeme und Kommunikation 72

4.1.2 Dynamische Speicherverwaltung 72

4.1.3 Persistenz 73

4.1.4 Modularisierung und Sichtbarkeit 73

4.1.5 Ausnahmebehandlung 75

4.2 Klassifizieren der Struktur und des Verhaltens von Objekten 75

4.2.1 Struktur von Objekten 76

4.2.2 Verhalten von Objekten 80

4.2.3 Allozieren und Initialisieren von Objekten 83

4.2.4 Lesen und Schreiben von Feldinhalten 85
4.3 Spezialisieren und Generalisieren von Objektklassen 85
4.3.1 Einfache Vererbung 86
4.3.2 Multiple Vererbung 90
4.3.3 Mixin-Vererbung 93
4.4 Reflektion und Spracherweiterungsprotokolle 97
4.4.1 Metaobjektklassen 97
4.4.2 Introspektionsprotokoll 98
4.4.3 Initialisierung von Klassen und Feldern 100
4.4.4 Vererbungsprotokoll 102
4.4.5 Feldzugriffsprotokoll 104
4.4.6 Allokations- und Initialisierungsprotokoll 107
4.4.7 Initialisierung generischer Operationen und Methoden 109
4.4.8 Methodenauswahl und generischer Dispatch 110
4.5 Abschlußeigenschaften der Sprachkonstrukte 111
4.5.1 Abschlußeigenschaften von Bindungen 112
4.5.2 Abschlußeigenschaften von Objekten 113
4.6 Zusammenfassung 114

5 Implementierungstechniken und Optimierungen 117
5.1 Überblick 117
5.1.1 Interpretation vs. Kompilation 117
5.1.2 Schichtenmodell vs. Einbettungsmodell 119
5.1.3 Inkrementelle, Modul- und Komplettkompilation 120
5.1.4 Laufzeitsystem vs. Entwicklungssystem 123
5.1.5 Optimierungen zur Übersetzungs-, Lade- und Laufzeit 123
5.2 Voraussetzungen für die Einbettungs- und Erweiterungstechnik 126
5.2.1 Syntaktische Erweiterbarkeit: Makros 126
5.2.2 Erweiterbarkeit der Datentypen 128
5.2.3 Zirkuläre Datenstrukturen 129
5.2.4 Funktionen höherer Ordnung 129
5.2.5 Closures 129
5.2.6 Interpretierer und Inkrementeller Compiler oder eval und compile . . 130
5.3 Systemarchitektur und Bootstrapping 131
5.4 Repräsentation von Objekten 132

5.4.1 Statische Objektrepräsentation 133
5.4.2 Funktionale Objektrepräsentation 136
5.5 Realisierung generischer Operationen 137
5.5.1 Generischer Dispatch 139
5.6 Sparsamer Speicherverbrauch 143
5.7 Vermeidung der Quellcode-Interpretation 145
5.7.1 Abstimmung syntaktischer und funktionaler Sprachkonstrukte . . . 146
5.8 Zusammenfassung 149

6 Der Einstieg: MCS nach dem Modell des Nachrichtenaustauschs 151
6.1 Diskussion von Flavors 152
6.1.1 BABYLON-Schnittstelle zu Flavors 152
6.1.2 Defizite vorhandener Objektsysteme 152
6.2 Entwurf von MCS 0.5 154
6.2.1 Ziele, Systemarchitektur und Vorgehensweise 154
6.2.2 Sprachkonstrukte der Objektebene 155
6.2.3 Sprachkonstrukte der Metaobjektebene 161
6.2.4 Hilfsmittel der Programmentwicklungsumgebung 163
6.3 Implementierung von MCS 0.5 164
6.3.1 Schritt 1: Objektrepräsentation und elementare Operationen 164
6.3.2 Schritt 2 und 4: syntaktische Konstrukte 171
6.3.3 Schritt 3: Bootstrapping des Kerns 173
6.3.4 Schritt 5: Vervollständigung des Metaobjektprotokolls 175
6.3.5 Realisierte Optimierungen 176
6.4 Performanzmessungen 176
6.4.1 Erster Vergleichstest 177
6.4.2 Zweiter Vergleichstest 179
6.5 Reale komplexe Anwendung: BABYLON 2.1 183
6.5.1 Implementierung von BABYLON-Flavors 183
6.5.2 Performanzmessungen 187
6.6 Bewertung von MCS 0.5 190
6.6.1 Sprachdesign 190
6.6.2 Implementierung 191
6.6.3 Zusammenfassung 192

7 Die Weiterentwicklung: MCS als kompatible Verbesserung von CLOS 193
7.1 Diskussion von CLOS . . . 194
7.1.1 Erzeugung und Initialisierung von Objekten in CLOS . . . 194
7.1.2 Slotzugriffe in CLOS . . . 195
7.2 Entwurf und Implementierung von MCS 1.0 . . . 196
7.2.1 Vereinfachungen gegenüber CLOS . . . 197
7.2.2 Sprachkonstrukte der Objektebene . . . 197
7.2.3 Vereinfachung der Erzeugung und Initialisierung von Objekten . . . 200
7.2.4 Vereinfachung der Slotzugriffe . . . 202
7.2.5 Mixin-Vererbung im Objektsystem-Kern . . . 203
7.2.6 Redefinieren und Reklassifizieren als Erweiterung . . . 208
7.3 Performanz von MCS 1.0 . . . 211
7.4 Bewertung . . . 216

8 Die Reflektion: Entwurf und Implementierung von TELOS 219
8.1 EuLisp und TELOS im Überblick . . . 221
8.2 Entwurf und Implementierung von TELOS 1.0 . . . 222
8.2.1 Ziele und Vorgehensweise . . . 223
8.2.2 Modularisierung von TELOS . . . 224
8.2.3 Objektebene . . . 226
8.2.4 Metaobjektebene . . . 236
8.2.5 Unterschiede zwischen TELOS und CELOS . . . 245
8.2.6 Performanzmessungen . . . 246
8.3 Exemplarische Spracherweiterungen . . . 257
8.3.1 Mixin-Vererbung . . . 257
8.3.2 Redefinieren von Klassen . . . 262
8.3.3 Dynamisches Klassifizieren von Objekten . . . 264
8.3.4 Methodenkombination . . . 265
8.3.5 Wissensrepräsentation mit Konzepten und Relationen: TINA . . . 271
8.4 Vergleich von EuLisp mit Java . . . 271
8.4.1 Klassen vs. Module . . . 272
8.4.2 Vergleich der Objektebene . . . 273
8.4.3 Vergleich der Metaobjektebene . . . 276
8.4.4 Abschlußeigenschaften . . . 277
8.5 Bewertung von TELOS . . . 277

9 Zusammenfassung und Ausblick **281**

A Glossar **287**

B Performanzmessungen von MCS 0.5 **293**

B.1 Erster Vergleichstest 293

B.2 Zweiter Vergleichstest 293

B.3 Vergleichstest für BABYLON-Wissensbasen 295

C Performanzmessungen von MCS 1.0 **297**

D Performanzmessungen von TELOS **299**

D.1 Vergleichstest auf Sun Workstation 299

D.2 Vergleichstest auf Apple Macintosh 299

E Implementierung der Mixin-Vererbung in TELOS **305**

Literaturverzeichnis **311**

Index **325**

Abbildungsverzeichnis

1.1 Abstammungsbaum funktionaler, imperativer und objektorientierter Programmiersprachen. 3

2.1 Architektur von BABYLON 10
2.2 Wechselwirkungen zwischen den Kriterien. 24

3.1 Reflektiver Kern in OBJTALK und ObjVlisp. 52
3.2 Methodenkombination in CLOS. 58
3.3 Konzeptuelle Verwandschaft objektorientierter Programmiersprachen. . . . 67

4.1 Einfache Vererbung und Methoden-Dispatch 89
4.2 Multiple und Mixin-Vererbung 95
4.3 Aufrufstruktur der Klasseninitialisierung 102
4.4 Berechnung der Feldzugriffsoperationen 105
4.5 Berechnung der Feldzugriffsoperationen bei einfacher Vererbung 106
4.6 Berechnung der Feldzugriffsoperationen bei multipler Vererbung 107
4.7 Abstrakte Allokationsoperationen 108
4.8 Primitive Allokationsoperationen 108
4.9 Operationen auf generischen Operationen und Methoden 111
4.10 Sprachaspekte im Überblick. 114

5.1 Inkrementelle Funktionskompilation 120
5.2 Datei-Kompilation 120
5.3 Modul-Kompilation 121
5.4 Applikations-Kompilation 122
5.5 Just-In-Time-Kompilation 124
5.6 Statische Objektrepräsentation 134

6.1 Beispiel der Verwendung von FLAVORS-Konstrukten in BABYLON. 153
6.2 Syntax der Klassen-Definition in MCS 0.5. 157

6.3 Syntax der Methodendefinition in MCS 0.5. 158
6.4 Syntax der Instanz-Erzeugung in MCS 0.5. 159
6.5 1. Vergleich der Ausführungszeiten auf Apple Macintosh II in Allegro Common Lisp 1.2.1 in Sekunden, Teil I. 178
6.6 1. Vergleich der Ausführungszeiten auf Apple Macintosh II in Allegro Common Lisp 1.2.1 in Sekunden, Teil II 178
6.7 Vergleich des Speicherbedarfs von Objektsystemen. 179
6.8 2. Vergleich der Ausführungszeiten auf Apple Macintosh II in Allegro Common Lisp 1.2.1 in Sekunden, Teil 1. 180
6.9 2. Vergleich der Ausführungszeiten auf Apple Macintosh II in Allegro Common Lisp 1.2.1 in Sekunden, Teil 2. 181
6.10 Vergleich der Ausführungszeiten auf SUN Workstation in Lucid Common Lisp in Sekunden, Teil I. 181
6.11 Vergleich der Ausführungszeiten auf SUN Workstation in Lucid Common Lisp in Sekunden, Teil II. 182
6.12 Vergleich der Ausführungszeiten des BABYLON-Frame-Prozessors in Sekunden. 188
6.13 Vergleich der Ausführungszeiten des BABYLON-Regel-Prozessors in Sekunden. 188
6.14 BABYLON-Prolog-Prozessor im Vergleich zur WAM in Sekunden. 189
6.15 BABYLON-Prolog-Prozessor im Vergleich zur WAM in lips. 190

7.1 Klassifikation der Metaklassen mit Hilfe der Mixin-Vererbung im Überblick. 204
7.2 Klassenhierarchie der primitiven Klassen in MCS. 205
7.3 Differenzierung zwischen primitiven und definierten Klassen. 206
7.4 Differenzierung zwischen abstrakten und instanziierbaren Klassen. 207
7.5 Differenzierung zwischen einfacher und multipler Vererbungsstrategie. . . . 208
7.6 Klassifikation redefinierbarer Klassen mit Hilfe der Mixin-Vererbung im Überblick. 209
7.7 Klassenhierarchie redefinierbarer Klassen. 210
7.8 Weitere Metaobjektklassen in MCS 1.0. 210
7.9 Umfang des Quellcodes von babylon in seiner Entwicklung. 211
7.10 Ausführungszeiten der Klassen- (**t1**) und Methodendefinition (**t2**) sowie der Instanzerzeugung (**t3**) in PCL, Franz Allegro CLOS und MCS 1.0 in Millisekunden. 213
7.11 Ausführungszeiten der Slotzugriffe (**t4**, **t5** und **t6**) in PCL, Franz Allegro CLOS und MCS 1.0 in Millisekunden. 213
7.12 Ausführungszeiten des generischen Dispatchs (**t7** und **t8**) in PCL, Franz Allegro CLOS und MCS 1.0 in Millisekunden. 214

7.13 Ausführungszeiten der Methodenkombination (**t5a**, **t5b**, **t5c** und **t5d**) in PCL, Franz Allegro CLOS und MCS 1.0 in Millisekunden. 215

7.14 Summe der Ausführungszeiten aller Tests in PCL, Franz Allegro CLOS und MCS 1.0 in Millisekunden. 216

8.1 Schichten-Architektur von ΤΕΛΟΣ. 224

8.2 Modularisierung der Metaobjektebene von ΤΕΛΟΣ. 226

8.3 Klassenhierarchie der Objektebene. 227

8.4 Syntax der Klassendefinition auf der Objektebene. 229

8.5 Syntax der Definition generischer Funktionen auf der Objektebene. 233

8.6 Syntax der Methodendefinition auf der Objektebene. 234

8.7 Allokations- und Initialisierungsoperationen der Objektebene. 235

8.8 Klassenhierarchie in EuLisp 237

8.9 Introspektionsoperationen für Klassen und Felder. 238

8.10 Introspektionsoperationen für generische Funktionen und Methoden. 238

8.11 Initialisierungsschlüsselwörter der Klasse `<class>`. 239

8.12 Initialisierungsschlüsselwörter der Klasse `<slot>`. 239

8.13 Initialisierungsschlüsselwörter der Klasse `<generic-function>`. 240

8.14 Syntax der Klassendefinition in EuLisp 241

8.15 Syntax der Definition generischer Funktionen auf der Metaobjektebene. . . 242

8.16 Syntax der Methodendefinition auf der Metaobjektebene. 242

8.17 Allokations- und Initialisierungsoperationen der Metaobjektebene. 243

8.18 Statische und dynamische Objekterzeugung **t1**, **t2**, **t3** und **t3a**: Vergleich der Ausführungszeiten von CLOS und ΤΕΛΟΣ auf einer Sun Workstation in Franz Allegro CL 4.3.1 in Millisekunden. 249

8.19 Statische und dynamische Objekterzeugung **t1**, **t2**, **t3** und **t3a**: Vergleich des Speicherbedarfs von CLOS und ΤΕΛΟΣ in Franz Allegro CL 4.3 auf Sun Workstation in Cons-Zellen und Bytes. 249

8.20 Statische und dynamische Feldzugriffe **t4**, **t4a**, **t5** und **t6**: Vergleich der Ausführungszeiten von CLOS und ΤΕΛΟΣ in Franz Allegro CL 4.3 auf Sun Workstation in Millisekunden. 250

8.21 Statische und dynamische Feldzugriffe **t4**, **t4a**, **t5** und **t6**: Vergleich des Speicherbedarfs von CLOS und ΤΕΛΟΣ in Franz Allegro CL 4.3 auf Sun Workstation in Cons-Zellen und Bytes. 251

8.22 Generischer Dispatch **t7**, **t8**, **t9**: Vergleich der Ausführungszeiten von CLOS und ΤΕΛΟΣ in Franz Allegro CL 4.3 auf Sun Workstation in Millisekunden. 251

8.23 Generischer Dispatch **t7**, **t8**, **t9**: Vergleich des Speicherbedarfs von CLOS und ΤΕΛΟΣ in Franz Allegro CL 4.3 auf Sun Workstation in Cons-Zellen und Bytes. 252

8.24 Methodenkombination **t8a-d**: Vergleich der Ausführungszeiten von CLOS und ΤΕΛΟΣ in Franz Allegro CL 4.3 auf Sun Workstation in Millisekunden. 253

8.25 Methodenkombination **t8a-d**: Vergleich des Speicherbedarfs von CLOS und ΤΕΛΟΣ in Franz Allegro CL 4.3 auf Sun Workstation in Cons-Zellen und Bytes. 254

8.26 Vergleich der Gesamtausführungszeiten von **t1**, **t2**, **t3**, **t3a**, **t5**, **t6**, **t8**, **t9** und **t8a-d** für CLOS und ΤΕΛΟΣ in Franz Allegro CL 4.3 auf Sun Workstation in Millisekunden. 255

8.27 Vergleich des Gesamtspeicherbedarfs von **t1**, **t2**, **t3**, **t3a**, **t5**, **t6**, **t8**, **t9** und **t8a-d** für CLOS und ΤΕΛΟΣ in Franz Allegro CL 4.3 auf Sun Workstation in Cons-Zellen und Bytes. 256

8.28 Vergleich der Gesamtausführungszeiten von **t1**, **t2**, **t3**, **t3a**, **t5**, **t6**, **t8**, **t9** und **t8a-d** für CLOS, MCS 0.5, MCS 1.0 und ΤΕΛΟΣ in Macintosh CL 4.1 auf einem Apple Macintosh PB280 in Millisekunden. 258

8.29 Vergleich des Gesamtspeicherbedarfs von **t1**, **t2**, **t3**, **t3a**, **t5**, **t6**, **t8**, **t9** und **t8a-d** für CLOS und ΤΕΛΟΣ in Macintosh CL 4.1 auf einem Apple Macintosh PB280 in Bytes. 258

8.30 Potentiell zu spezialisierende Operationen des Metaobjektprotokolls 260

8.31 Tatsächlich zu spezialisierende Operationen des Metaobjektprotokolls . . . 261

8.32 Modell der einfachen Methodenkombination mit dem Primitiv `super` aus Smalltalk. 265

8.33 Modell der einfachen Methodenkombination mit dem Primitiv `inner` aus BETA. 266

8.34 Modell der Standardmethodenkombination aus CLOS. 266

8.35 Beispiele aus Java 275

Tabellenverzeichnis

3.1 Vergleich der Sprachen anhand der Designkriterien. 68

3.2 Vergleich der bereitgestellten Konzepte objektorientierter Programmiersprachen. 70

8.1 Vergleich der Sprachen anhand der Designkriterien. 279

8.2 Vergleich der Sprachaspekte. 280

B.1 Ausführungszeiten auf Mac II, Allegro CL in Sekunden 293

B.2 Ausführungszeiten auf Macintosh II, Allegro CL in Sekunden 294

B.3 Ausführungszeiten auf SUN, Lucid CL in Sekunden 294

B.4 Vergleich der Ausführungszeiten von BABYLON-Prozessoren in Sekunden . 295

B.5 BABYLON-Prolog-Prozessor im Vergleich zur WAM in Sekunden. 295

B.6 BABYLON-Prolog-Prozessor im Vergleich zur WAM in lips. 295

C.1 Ausführungszeiten auf Sun Workstation, Franz Allegro Common Lisp (ACL) in Millisekunden 297

C.2 Speicherbedarf von PCL und MCS in Franz Allegro CL in Megabytes . . . 298

D.1 Ausführungszeiten auf einer Sun Workstation in Franz Allegro CL 4.3.1 in Millisekunden. 300

D.2 Speicherbedarf auf einer Sun Workstation, Franz Allegro CL 4.3.1 in der Anzahl der cons-Zellen und weiteren Bytes. 301

D.3 Ausführungszeiten auf einem Apple Macintosh PB 280 in Macintosh CL 4.1 in Millisekunden. 302

D.4 Speicherbedarf auf einem Apple Macintosh PB 280 in Macintosh CL 4.1 in Bytes. 303

Kapitel 1

Einleitung

> *A complex system that works is invariably found to have evolved from a simple system that worked. ... A complex system designed from scratch never works and cannot be patched up to make it work. You have to start over, beginning with a working simple system.*
>
> Gall, J. 1986. Systemantics: How Systems Really Work and How They Fail. 2. Aufl., Ann Arbor, MI: The General Systemantics Press, S. 65. [Booch, 1991].

1.1 Motivation

Objektorientierte Sprachkonstrukte kann man inzwischen in fast allen Programmiersprachen finden. Leider gibt es immer noch keinen Konsens darüber, was ihre Essenz ausmacht, welche Aufgaben sie unterstützen sollen und wie sie gegenüber anderen Paradigmen abgegrenzt werden können. Bei genauerem Hinsehen stellt man schnell fest, daß es keine einheitliche Terminologie objektorientierter Sprachkonstrukte gibt. Für gleiche Konzepte werden unterschiedliche Begriffe verwendet und dieselben Wörter bezeichnen unterschiedliche Konzepte in verschiedenen Programmiersprachen.

Historisch gesehen kann man auch bei objektorientierten Programmiersprachen zwei Entwicklungsstränge identifizieren. Der eine Strang war vom Bemühen um problemadäquate Ausdrucksmittel geprägt, oft unter Vernachlässigung des Performanzaspekts. Der andere Strang stellte Effizienzfragen in den Vordergrund und ließ Erkenntnisse und Anforderungen aus der objektorientierten Wissensrepräsentation unberücksichtigt. Beispielsweise würde man Smalltalk dem ersten Strang zuordnen, während C++ eindeutig zum zweiten Strang gehört.

In Lisp, das klar dem ersten Strang zuzuordnen ist, haben Objektsysteme eine lange Tradition, weil Lisp besonders einfach erweiterbar ist und nahezu uneingeschränkte Möglichkeiten zur Programm- und Datenabstraktion bietet. Dadurch konnte man mit Lisp als Basissprache sehr schnell ein breites Spektrum objektorientierter Konzepte, bis hin zu mächtigen Wissensrepräsentationssprachen, erforschen und praktisch validieren. Den Status eines Standards hat hier CLOS als Nachfolger der Systeme LOOPS und FLAVORS

erreicht. Das innovative an CLOS im Vergleich zum Smalltalk-ähnlichen FLAVORS ist die bessere Integration objektorientierter und funktionaler Sprachmittel sowie sein Spracherweiterungsprotokoll, auch *Metaobjektprotokoll* genannt. Beim Entwurf von CLOS wurde jedoch zu wenig auf die semantische Klarheit und die Orthogonalität der Sprachkonstrukte geachtet, was einen schlechten Programmierstil begünstigt und zu schwer verständlichen Programmen führen kann. Die Funktionalität von CLOS, wie auch von COMMONLISP insgesamt, wurde zu stark auf die Programmentwicklung ausgerichtet, so daß keine sicheren statischen Programmanalysen und nur wenige globale Optimierungen durch Compiler möglich sind. Im Vergleich zu objektorientierten Programmiersprachen des zweiten Strangs wie C++, die viel leichter traditionelle Compilerbautechniken einsetzen können, führt dies im Ergebnis zu inhärenter Ineffizienz der Anwendungen. Laufzeit kann nur auf Kosten der Robustheit verbessert werden und der Speicherbedarf des Laufzeitsystems ist immens, weil es den Compiler und den Interpreter mitenthalten muß.

Auf der anderen Seite stellen Sprachen wie C++ für viele Anwendungsbereiche, wie z. B. wissensbasierte Systeme, keine akzeptable Alternative dar. Die Gründe hierfür sind vielfältig: es beginnt mit der fehlenden automatischen Speicherverwaltung, setzt sich fort mit dem Erbe aus C wie der Möglichkeit, Pointer zu manipulieren mit allen negativen Konsequenzen, und endet mit der unzureichenden Unterstützung des Spezialisierens und Generalisierens von Softwarebausteinen, des Hauptmerkmals objektorientierter Programmierung. Mit dem Boom des Internets und der einhergehenden Etablierung von JAVA gibt es zwar eine C++-ähnliche Alternative bzgl. der beiden erstgenannten Probleme. Weitere Anforderungen an eine problemorientierte Programmiersprache wie syntaktische und funktionale Erweiterbarkeit der Sprache werden jedoch auch von JAVA nicht erfüllt. Und auch die objektorientierten Konzepte von JAVA weisen deutliche Defizite auf. Insbesondere ist hier die Vermischung der Mittel zur Modularisierung mit denen zur Klassifizierung, das Shadowing von Instanzvariablen und der Methodendispatch zu nennen. Auch wenn in der überarbeiteten Sprachversion JDK1.1 die reflektiven Sprachmittel deutlich erweitert wurden, ist ihre Spezialisierung jedoch nicht möglich. Das JAVA-Objektsystem ist somit nicht spezialisierbar und für die Einbettung höherer Modellierungssprachen ungeeignet.

Es stellt sich somit die Frage, ob die Lücke zwischen den beiden Strängen geschlossen und der Gegensatz zwischen Erfordernissen einer problemadäquaten und somit anwendungsnahen objektorientierten Sprache und den Restriktionen einer für die gegenwärtige Rechnergeneration effizient implementierbaren Sprache überwunden werden kann.

Als Antwort auf diese Frage wurde ein neuer Lisp-Dialekt EULISP entwickelt. EULISP verbindet traditionelle Stärken von Lisp als einer funktional-imperativen, syntaktisch erweiterbaren Sprache mit bewährten Konzepten anderer Programmiersprachen. Es bietet ein Modulkonzept und weist selbst eine modulare Architektur auf. Es stellt einfache Mechanismen für nebenläufige und verteilte Berechnungen sowie ihre Synchronisation zur Verfügung. Ein Konzept für die Ausnahmebehandlung unterstützt die Realisierung ausfallsicherer Software. Sein Objektsystem TELOS, das ich als Resultat dieser Arbeit präsentiere, ist integraler Bestandteil der Sprache. Es greift die objektorientierten Grundideen aus CLOS auf und stellt eine vergleichbare Funktionalität unter Berücksichtigung des Verursacherprinzips zur Verfügung. Im Gegensatz zu CLOS enthält TELOS keine Sprachkonstrukte, die den Einsatz traditioneller Compilerbautechniken verhindern. Durch Modul- und Komplettkompilation kann für objektorientiert implementierte Applikationen in EULISP schließlich die gleiche Laufzeit- und Speichereffizienz erreicht werden, wie in C++.

Aber auch die portable Implementierung von ΤΕΛΟΣ in COMMONLISP weist eine deutlich bessere Performanz als CLOS auf. Abbildung 1.1 zeigt den Kontext dieser Arbeit im Abstammungsbaum funktionaler, imperativer und objektorientierter Programmiersprachen.

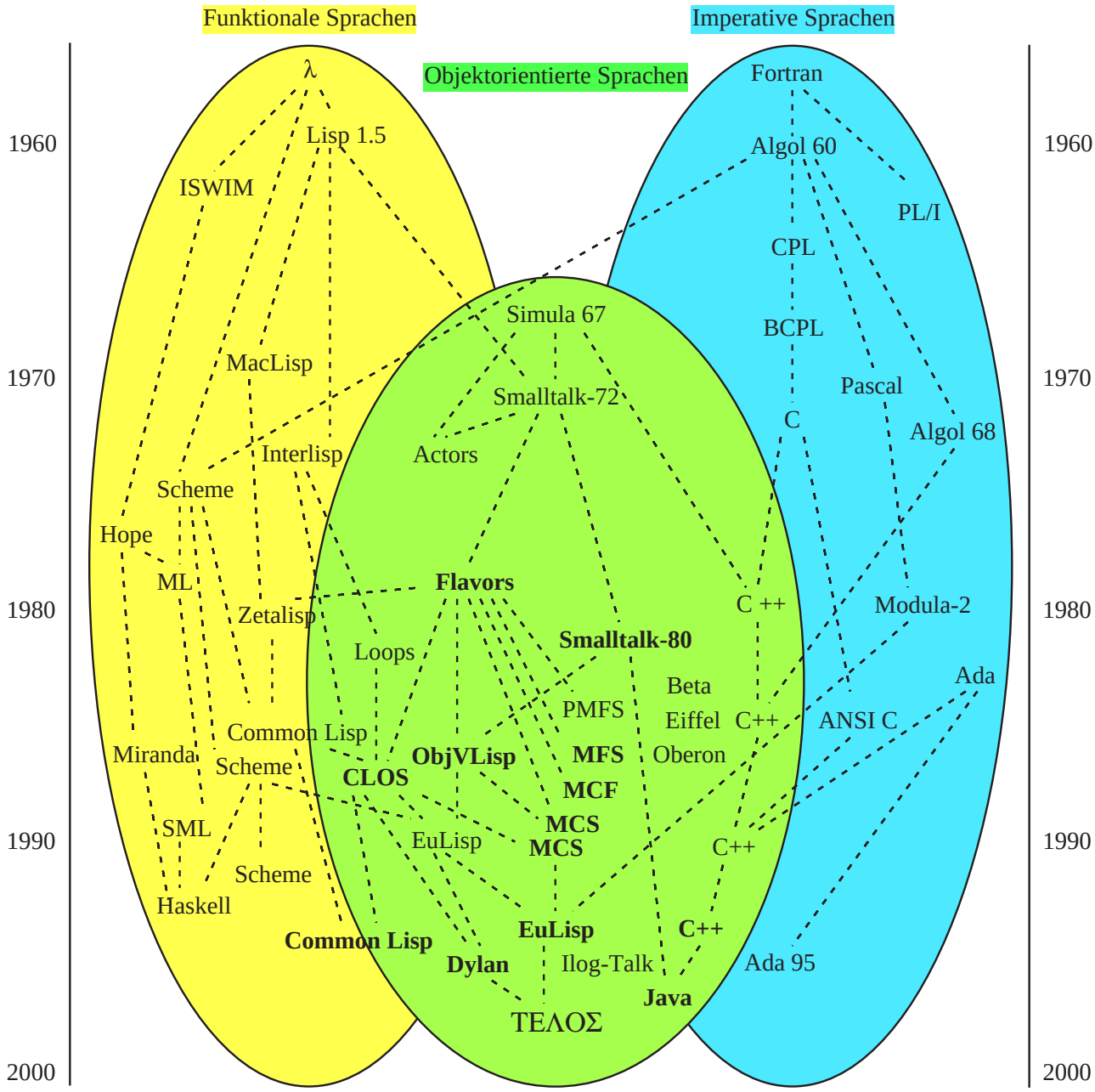


Abbildung 1.1: Abstammungsbaum funktionaler, imperativer und objektorientierter Programmiersprachen.

1.2 Ziel und Arbeitshypothese

Das Ziel dieser Arbeit ist der Entwurf von Objektsystemen für funktionale und imperative Programmiersprachen sowie ihre portable Implementierung als Erweiterung von Lisp, die den Anforderungen komplexer Anwendungen bezüglich konzeptueller Klarheit, Flexibilität und Effizienz standhalten. Ausgehend von FLAVORS und CLOS werde ich schrittweise zeigen, wie man die Effizienz von Objektsystemen verbessern kann, ohne die wesentlichen objektorientierten Konzepte aufzugeben. Allerdings gehe ich in meiner Arbeitshypothese davon aus, daß signifikanten Effizienzverbesserungen eine kritische Reflektion dieser Konzepte vorausgehen muß.

Ausgehend von den Anforderungen komplexer Anwendungen in ihrem Software-Lebenszyklus sind klare *Entwurfskriterien* für das Objektsystem zu erarbeiten. Es muß diskutiert werden, welche Analyse- Design- und Programmiertechniken durch konkrete Sprachkonstrukte unterstützt werden sollen. Als das wichtigste Konzept objektorientierter Sprachen sehe ich die *Vererbung* an. Hierfür soll eine Lösung gefunden werden, die das *Spezialisieren und Generalisieren* von Softwarebausteinen unterstützt und den Gegensatz zwischen *einfacher und multipler Vererbung* überwindet.

Als weitere Annahme meines Ansatzes lege ich die Erkenntnis zugrunde, daß problemorientierte Sprachen nicht ein für alle mal und für alle Problemklassen entworfen werden können. Mit dem Versuch, ein vollständiges und abgeschlossenes Objektsystem zu entwerfen, können die gesetzten Ziele daher nicht erreicht werden. Vielmehr kommt es darauf an, einen einfachen und effizienten Objektsytemkern zu konstruieren, der für die jeweiligen Bedürfnisse spezialisiert werden kann. Dies ist die Grundidee eines *Spracherweiterungsprotokolls* (engl. *metaobject protocol*).

Bisherige Spracherweiterungsprotokolle wie das CLOS MOP verhinderten aber die Komplettkompilation sie nutzender Anwendungen. In dieser Arbeit soll gezeigt werden, wie man Spracherweiterungsprotokolle so entwirft, daß sie mit der *Komplettkompilation* verträglich sind und deshalb die Performanz komplexer Anwendungen signifikant verbessern.

Den experimentellen Rahmen dieser Arbeit bildet die Familie der Lisp-Sprachen. Vor dem Hintergrund konkreter Anforderungen einer realen Anwendung, der KI-Werkbank BABYLON in ihrer Entwicklung vom Forschungsprototypen zum kommerziellen Produkt babylon, habe ich zwei Objektsysteme für COMMONLISP entworfen und implementiert: MCS 0.5 und MCS 1.0. Aus den gewonnenen Erfahrungen und Erkenntnissen wurde unter meiner Federführung das Objektsystem TELOS für EULISP definiert. Dabei mußte keine Rücksicht auf die Kompatibilität mit einer existierenden Sprache wie bei MCS, CLOS oder auch bei C++ genommen werden. Daher spiegeln die Sprachkonstrukte von EULISP die von mir erarbeiteten objektorientierten Konzepte am besten wider. Diese Konzepte sind das wichtigste und auf andere funktionale und imperative Sprachen übertragbare Ergebnis, das ich im Vergleich mit JAVA diskutieren werden. Ohne den Fundus an durchgeführten Experimenten hätte es jedoch nicht erzielt werden können.

Schließlich muß eine vielleicht trivial erscheinende Grundhypothese dieser Arbeit genannt werden, die offensichtlich nicht von allen geteilt wird, wenn man die Spezifikation von CLOS betrachtet. Es ist die Überzeugung, daß einfache Sprachmittel letztlich besser als komplizierte sind, weil sie das Software-Engineering von der Analyse bis zur Wartung besser unterstützen und erleichtern. Nur wenn die Grundbausteine und -mechanismen

von Programmiersprachen einfach sind, kann die inhärente Komplexität von Software-Systemen beherrschbar werden. Damit verbunden ist das *Verursacherprinzip:* aufwendige Funktionalität, die nicht in Anspruch genommen wird, darf auch nichts kosten. Und wenn sie in Anspruch genommen wird, sollen auch nur an dieser Stelle höhere Kosten entstehen. Dies wird auch als die Forderung nach einem für Programmierer *transparenten Effizienzmodell* der Sprachkonstrukte bezeichnet. Das Verursacherprinzip soll unter anderem am Beispiel der aus CLOS bekannten Redefinitionsmöglichkeiten demonstriert werden.

1.3 Hintergrund und Aufbau der Arbeit

Diese Arbeit ist im Kontext meiner Projekttätigkeit im Forschungsbereich Künstliche Intelligenz der GMD - Forschungszentrum Informationstechnik GmbH entstanden. Gerade beim Design einer Programmiersprache sorgt die unmittelbare Umgebung, in der der Bedarf an objektorientierten Sprachkonstrukten entsteht, für ihre bewußte Ausrichtung auf die praktisch relevanten Fragen. Damit ist auch klar, welche Klasse von Anwendungen im Fokus meiner Arbeit stand: Im Unterschied zu C++ mit seiner Nähe zu Betriebssystementwicklungen waren es hier Systeme der Künstliche Intelligenz. Die unterschiedlichen Resultate dürfen daher nicht verwundern.

Daß Lisp eine starke Tradition in der Künstliche Intelligenz Forschung hat, ist hinlänglich bekannt. Die meisten größeren KI-Systeme wurden in Lisp implementiert. Als die auf Lispmaschinen entwickelte KI-Werkbank BABYLON auf andere Rechner- und Betriebssystemplattformen portiert werden sollte, stellte sich die Frage nach einem portablen und effizienten Objektsystem, das mit FLAVORS kompatibel war. Bei der späteren Entwicklung des kommerziellen Produkts babylon war dann die Kompatibilität mit CLOS gefragt. Mit MCS 0.5 gelang es mir, die Laufzeiteffizienz von BABYLON gegenüber vorher benutzten Objektsystemen wie WFS und MCF bis zu einem Faktor 100 zu verbessern, und zwar bei gleichzeitig geringerem Speicherbedarf. Die Verbesserung betraf sowohl die Entwicklungs- als auch die Ausführungszeiten von BABYLON-Wissensbasen. Im nächsten Schritt wurde eine zu CLOS kompatible Version von MCS realisiert, die dann auch im Produktprojekt babylon eingesetzt wurde, weil sie effizienter als verfügbare kommerzielle Systeme war.

Im Rahmen des APPLY-Projekts habe ich später intensiv mit Kollegen aus dem Compilerbau der Universität Kiel und der FhG ISST Berlin zusammengearbeitet. Dabei ging es um die Komplettkompilation von Lisp [Bretthauer *et al.*, 1994].

Neben der Entwicklung von MCS habe ich mich intensiv an der Definition von EULISP beteiligt. Unter meiner Federführung und unter Beteiligung weiterer Kollegen in der *EuLisp Working Group* konnte eine überarbeitete Version von TELOS als Teil von EULISP 0.99 spezifiziert und in [Bretthauer *et al.*, 1992] sowie [Padget *et al.*, 1993] veröffentlicht werden.

Meine Referenzimplementierung von TELOS in COMMONLISP, genannt CELOS, diente als Implementierungsbasis für das Wissensrepräsentationssystem TINA. Jürgen Kopp hat in seiner Dissertation [1996a] die Vorteile der Realisierung von TINA durch Wiederverwenden und Erweitern von TELOS ausführlich dargestellt. Die von EULISP bereitgestellten Sprachmittel wurden dabei theoretisch und praktisch überprüft und bestätigt.

Meine Arbeit ist in zwei Teile gegliedert. Der erste Teil enthält diese Einleitung, das Szenario, den Stand der Technik und die entwickelten Konzepte für den Entwurf und die

Implementierung effizienter Objektsysteme. Dies sind die Kapitel 1 bis 5. Danach kommt der experimentelle Teil der Arbeit, in dem drei konkrete Objektsysteme mit entsprechenden Performanzmessungen vorgestellt werden (Kapitel 6 bis 8). Abschließend folgt eine Zusammenfassung der Ergebnisse und der Ausblick auf künftige Arbeiten auf diesem Gebiet.

Nun könnte die Frage aufkommen, warum es drei Objektsysteme sein müssen? Kann man die relevanten Ergebnisse nicht schon mit einem Objektsystem zeigen? Und worin unterscheiden sich die drei Systeme?

Ich habe mich für diese Darstellung entschieden, weil sie den tatsächlichen Weg vom Problem zum Lösungsansatz und schließlich zum Endergebnis am besten trifft. Das Design und die Implementierung effizienter Objektsysteme können am besten als ein inkrementeller Prozeß verstanden werden. Dabei habe ich mich bemüht, Redundanzen zu vermeiden und gleichzeitig jedes Kapitel als eigenständige Einheit lesbar zu machen.

Kapitel 2

Wie entwirft man effiziente Objektsysteme?

> *I will assert that the hardest part in any design is deciding what it is you want to design.*
>
> Frederick P. Brooks, Jr. 1993. Keynote address: Language Design as Design, The Second History of Programming Languages Conference (HOPL-II), [Brooks, 1996].

Den übergeordneten Rahmen für diese Arbeit bilden der Entwurf und die Implementierung von universellen höheren Programmiersprachen im Hinblick auf die Anforderungen komplexer, insbesondere wissensbasierter Systeme. Performanz ist dabei ein besonders wichtiges Kriterium. Das Komplexitätsproblem stellt sich hier nicht nur im engeren algorithmischen Sinn, vielmehr ist es umfassend im Blick auf die Aufgaben der Software-Entwickler zu verstehen. Entsprechend ist auch Effizienz ein vielschichtiges Ziel. Ich gehe von der Grundannahme aus, daß universelle Sprachen, die nicht nur *einen* Programmierstil unterstützen, eine deutlich höhere praktische Relevanz für wissensbasierte Systeme sowie für komplexe Anwendungen generell besitzen, als puristische Ansätze, die primär zu einer fundierten Theoriebildung beitragen.

Im Mittelpunkt meiner Betrachtungen stehen die objektorientierten Sprachanteile einer Programmiersprache, die man zusammenfassend auch als *das Objektsystem* einer Programmiersprache bezeichnet. Dazu orthogonale Sprachaspekte müssen natürlich auch berücksichtigt und aufeinander abgestimmt werden; sie bilden jedoch nicht den Schwerpunkt dieser Arbeit. Die generellen Fragen des Designs und der Implementierung objektorientierter Sprachmittel werden für die Familie der Lisp-Sprachen konkretisiert. Die gewonnenen Erkenntnisse gelten jedoch generell für funktionale und imperative Programmiersprachen. Lisp habe ich gewählt, weil es als eine problemorientierte Sprache immer noch eine herausragende Stellung bei der Konstruktion wissensbasierter Systeme hat. Andere Sprachen, die traditionell eher als maschinenorientiert zu bezeichnen sind, wie C und C++, erfüllen von ihren Ausdrucksmöglichkeiten her nur unzureichend die Anforderungen dieser Anwendungsklasse. Ausgehend von CommonLisp mit Flavors und CLOS wird eine Alternative für EuLisp entwickelt, die einfacher, zuverlässiger und effizienter implementierbar ist. Abschließend wird die Programmiersprache Java an den hier gesetzten Entwurfskriterien

gemessen und mit EuLisp verglichen.

Will man Sprachkonstrukte zielgerichtet entwickeln und ein optimales Kosten-/Nutzenverhältnis erreichen, so muß berücksichtigt werden, daß sie von Programmierern in einem Handlungskontext verwendet werden. Dieser bestimmt die Kriterien und Leitlinien für ihren Entwurf und ihre Implementierung. Jede Phase im Software-Lebenszyklus komplexer Anwendungen muß ihre besondere Berücksichtigung finden.

Die Erfahrungen mit der Entwicklung und dem Einsatz von Software lehren, daß die Beherrschung der Komplexität ein zentrales Problem darstellt. Objektorientierte Sprachmittel sind ein Beitrag zur Lösung dieses Problems, sie sind der Gegenstand dieser Arbeit. Natürlich sind sie nicht das Allheilmittel, um das ganze Problem zu lösen, wie es auch andere Ansätze, wie z.B. CASE, nicht sind.

Die Wiederverwendung von Software ist ein vielzitiertes Schlagwort bei der Suche nach Wegen aus der sogenannten Software-Krise [Wirth, 1994]. Diese Arbeit zeigt, welchen Beitrag objektorientierte Sprachmittel zur Wiederverwendbarkeit leisten können, wo ihre Grenzen diesbezüglich liegen und welche Aspekte an anderer Stelle, z.B. von Betriebssystemen oder sogenannter *middleware*, aufgegriffen und gelöst werden müssen. Eine objektorientierte Sprache sollte so entworfen werden, daß nicht nur damit realisierte Anwendungen wiederverwendet werden können, sondern auch die Programmiersprache selbst bzw. ihr Objektsystem. Dazu müssen zunächst immer wiederkehrende Aufgabenstellungen bei der Realisierung komplexer Anwendungen identifiziert werden. Dann können entsprechende Lösungsmuster in einer Programmiersprache angeboten werden, die erweiterbar und an ein konkretes Problem anpaßbar sind.

Gesichtspunkte des Software-Engineering (SE) spielen bei den folgenden Diskussionen und Entwurfsentscheidungen eine wichtige Rolle, sind aber selbst nicht Gegenstand dieser Arbeit. Vielmehr werden die allgemeinen Grundsätze des Software-Engineering im konkreten Kontext pragmatisch angewandt. Dabei beziehe ich mich auf [Jalote, 1997], was die allgemeinen SE-Methoden angeht, und auf [Booch, 1994], wenn es um die speziellen objektorientierten SE-Methoden geht.

Im Zusammenhang mit der Komplexität von Anwendungen stellt sich auch immer wieder die Frage nach ihrer Performanz. Die Sprachmittel einer Programmiersprache garantieren noch nicht die Effizienz von Anwendungen, aber sie ermöglichen und unterstützen bzw. erschweren oder verhindern sie. Oft entscheidet auch nicht die prinzipielle, sondern die praktische Machbarkeit bzw. eine bestimmte Tradition des Compiler- und Interpreterbaus für eine Sprachfamilie über den Grad ihrer Speicher- und Laufzeiteffizienz.

Als Implementierungstechnik habe ich die portable und inkrementelle Spracherweiterung, auch Spracheinbettung genannt, gewählt. Dennoch werden die Techniken des traditionellen Compilerbaus den Sprachentwurf signifikant beeinflussen, weil sie wesentlich weiterreichende Programmoptimierungen erlauben. Insbesondere soll Modul- und Komplettkompilation ermöglicht und unterstützt werden. Das Ergebnis mündet in die Spezifikation von Τελος in EuLisp sowie in seine Referenzimplementierung Celos.

Um die Anwendungen zu charakterisieren, die von den hier entwickelten Lösungen profitieren sollen, stelle ich zunächst drei Beispiele vor. Anschließend werden die verschiedenen Phasen des Software-Lebenszyklus im Hinblick auf ihre Sprachanforderungen diskutiert. Daraus werden dann die Kriterien für objektorientierte Programmiersprachen abgeleitet, die später als Meßlatte zur Bewertung der Lösungen dienen werden.

2.1 Beispiele komplexer Softwaresysteme: BABYLON, TINA, KIKon

Als begleitendes Referenzbeispiel für diese Arbeit dient hauptsächlich die KI-Werkbank BABYLON. Sie liefert die experimentelle Basis für die Performanzmessungen und Vergleiche mit anderen Objektsystemen. Darüberhinaus wurde das Endergebnis CELOS zur Realisierung des Wissensrepräsentationssystems TINA verwendet. Als weiteres Beispiel stelle ich KIKONL vor, das der ressourcenorientierten Modellierung von Telekommunikationsdiensten und -komponenten dient.

Am Beispiel BABYLONwird der gesamte Softwarelebenszyklus einer komplexen Anwendung betrachtet. Alle drei Beispiele machen deutlich, welche Vorteile man erlangt, wenn eine objektorientierte Modellierungssprache als Erweiterung und Spezialisierung einer objektorientierten Programmiersprache realisiert werden kann. Darüberhinaus können natürlich weitere komplexe Softwaresysteme genannt werden, die von Ergebnissen dieser Arbeit profitieren können, z. B. Verbmobil [Wahlster, 1997].

2.1.1 KI-Werkbank BABYLON

Babylon ist ein Softwaresystem, mit dem wissensbasierte Anwendungen entwickelt und betrieben werden können. Es basiert auf der Grundidee von Franco di Primio, verschiedene Wissensrepräsentationsformalismen zu kombinieren, um beim Aufbau anwendungsspezifischer Wissensbasen möglichst adäquate Mittel verwenden zu können [di Primio, 1993].

Man machte die Erfahrung, daß schon für nur eine realistische Anwendung mehr als ein Formalismus notwendig war, um eine für den Modellierer kognitiv adäquate Lösung zu ermöglichen. Wie man die Schnittstellen zwischen den Formalismen wie Objekte (Frames), Logik und Produktionsregeln sinnvoll festlegt, war zunächst alles andere als trivial. Auf jeden Fall sollte der Wissensingenieur von diesem Problem entlastet werden. Die Herausforderung bestand darin, die Formalismen für den Wissensingenieur zu integrieren, um ihm eine uniforme Sicht auf seine Wissensbasis zu ermöglichen, aber gleichzeitig eine modulare Architektur und Realisierung des Systems beizubehalten. Die Lösung wird in Abbildung 2.1 dargestellt. Sie zeigt den Daten- und Kontrollfluß zwischen den drei Interpretierern, die den drei Formalismen entsprechen, und dem Metainterpretierer, der die Gesamtverantwortung für die Interpretation von Ausdrücken der Wissensbasis hat. Der Metainterpretierer delegiert die Interpretation von Teilausdrücken an den zuständigen Interpretierer. Kann dieser nicht den ganzen Teilausdruck bearbeiten, tut er so viel er kann und gibt den Rest wieder an den Metainterpretierer zurück. Dieser sorgt dann dafür, daß ein anderer Formalismus den restlichen Teil übernimmt, das entsprechende Teilergebnis berechnet und an den Metainterpretierer zurückliefert, der es dann weiterleitet - und so weiter bis alle Sprachausdrücke abgearbeitet sind.

Die Entwicklung von BABYLON[1] begann 1984 in der GMD, setzte sich fort in verschiedenen Verbundprojekten mit externen Partnern in der Forschung und in der Industrie und fand im veröffentlichten Forschungsprototypen[2] zunächst seinen Abschluß [Christaller *et al.*, 1989]. Anschließend wurde der Forschungsprototyp gemeinsam von der VW

[1] Diese Schreibweise verwende ich im folgenden für den Forschungsprototypen.
[2] ftp://ftp.gmd.de/GMD/ai-research/Software/Babylon/

Gedas GmbH und der GMD zu einem kommerziellen Produkt babylon[3] weiterentwickelt [VW-Gedas, 1991].

Die wissenschaftlich-technische Herausforderung bestand darin, eine integrierende und erweiterbare Architektur für unterschiedliche Wissensrepräsentationsformalismen zu finden. So wurde die Offenheit für neue Formalismen z. B. dadurch nachgewiesen, daß ein Constraint-System zu einem späteren Zeitpunkt hinzugefügt wurde, ohne daß bereits vorhandene Formalismen (Frames, Produktionsregeln und Prolog) geändert werden mußten. Aber auch bestehende Formalismen wurden um zusätzliche Funktionalität erweitert.

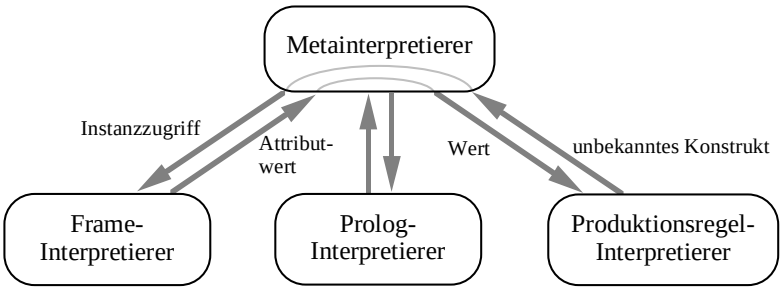


Abbildung 2.1: Architektur von BABYLON

Die softwaretechnische Aufgabe bestand darin, das Architekturkonzept so zu realisieren, daß das Gesamtsystem verständlich, wartbar, konfigurierbar, erweiterbar und portabel ist.

Allein an der softwaretechnischen Entwicklung des Forschungsprototypen waren über einen Zeitraum von 5 Jahren mit unterschiedlichem Zeitaufwand etwa 25 Personen beteiligt. Für die Produktentwicklung wurden in 2 Jahren ca. 30 Personenjahre aufgewendet. Diese Zahlen lassen an der Komplexität des Softwaresystems in seinem Lebenszyklus keinen Zweifel aufkommen.

Realisierung von BABYLON

BABYLON wurde von Anfang an objektorientiert implementiert. Zunächst in ZetaLisp auf Symbolics Lisp-Maschinen unter Verwendung von Flavors begonnen, wurde BABYLON bald auf CommonLisp portiert. Dabei beschränkte man sich bewußt auf solche Sprachkonstrukte, die einfach und problemlos portierbar waren. Da nicht alle CommonLisp-Systeme über eigene Realisierungen von Flavors verfügten, wurden die Objektsysteme WFS [de Buhr und Friederich, 1987] als Weiterentwicklung von MFS [Christaller, 1986], MCF [di Primio, 1988] und schließlich MCS verwendet. Dadurch wurde es möglich, BABYLON in verschiedenen Lisp-Dialekten[4], auf verschiedenen Rechnern[5] und unter verschiedenen Betriebssystemen[6] quellcode-identisch zu benutzen. Die objektorientierten Spracheigenschaften von CommonLisp und Flavors werde ich später ausführlich behandeln. Um die Relevanz eines effizienten Objektsystems anzudeuten, sei hier der Hinweis erlaubt, daß mit dem Übergang auf MCS der Speicherplatzbedarf von BABYLON

[3] Diese Schreibweise verwende ich im folgenden für das kommerzielle Produkt.

[4] Le-Lisp von ILOG; CommonLisp von Apple, DEC, Franz Inc., Gold Hill Computers, Lucid Inc., Symbolics Inc., Texas Instruments, Xerox PARC und von den Universitäten Texas und Carnegie Mellon

[5] Macintosh, VAX, IBM-AT, Cadmus 9000, Siemens MX-2, Sun, Symbolics, TI-Explorer

[6] MacOS, micro-VMS, MS-DOS, MUNIX, SINIX, UNIX

einschließlich seiner Applikationen um 30 Prozent verringert und die Laufzeiten der Formalismen um den Faktor 10 bis 100 verkürzt wurden. Sollte jetzt jemand meinen, die Performanz von Softwaresystemen spiele heute keine Rolle mehr, so möchte ich aus Erfahrung entgegnen, daß schon die jeweils aktuelle Systemsoftware den Leistungszuwachs der neuesten Hardware voll für sich beansprucht, und dadurch für Anwendungen nicht unbedingt eine Verbesserung eintritt.

Der Frame-Formalismus in BABYLON

Das FLAVORS-System diente nicht nur als Systemimplementierungssprache, sondern war auch der Ausgangspunkt für den objektorientierten Wissensrepräsentationsformalismus in BABYLON. Dieser Frame-Formalismus unterscheidet zwischen den eigentlichen Frames und ihren Instanzen. Entsprechend wurden Frame-Definitionen auf Flavor-Definitionen abgebildet und die Erzeugung von Frame-Instanzen auf die Erzeugung von Flavor-Instanzen. Das operationale Wissen über Frames wurde mit Behaviors modelliert, die auf Flavor-Methoden abgebildet wurden. Da FLAVORS kein Erweiterungsprotokoll bereitstellte, mußte die zusätzliche Funktionalität, z. B. komplexere Zugriffe auf Slots bzw. Instanzvariablen, relativ umständlich implementiert werden. Dies führte in der Folge zu einer Lösung, die auch konzeptuell erst in späteren Versionen verbessert werden konnte.

Von BABYLON 2.1 zu babylon 3.0

Bei dem Schritt zum kommerziellen Produkt wurde die Software komplett reimplementiert. Als Systemimplementierungssprache wurde wieder COMMONLISP gewählt. Statt FLAVORS entschied man sich aber für den angekündigten neuen Standard CLOS.

Die Funktionalität der Wissensrepräsentationsformalismen wurde im wesentlichen beibehalten, ihre konkrete Ausprägung und Aufteilung wurden aber gründlich überarbeitet. So wurde sowohl die Syntax der Wissensrepräsentationssprache als auch das Implementierungskonzept geändert. Neu hinzu kamen komfortable Entwicklungswerkzeuge für den Wissensingenieur.

Gleichzeitig war es aber aus Marketinggründen erforderlich, zumindest in einer Übergangszeit alte Wissensbasen mit dem neuen System verarbeiten zu können. Schon während der Entwicklung des kommerziellen Produkts mußte das halbfertige System in halbjährlichem Rhythmus auf Messen (CeBit und Systems) präsentiert werden. Man mußte also softwaretechnisch in der Lage sein, Teile des alten Systems in FLAVORS-Syntax und neue Softwarekomponenten in CLOS-Syntax in kürzester Zeit miteinander zu integrieren. So kompliziert das war, so einfach war die Lösung mit Hilfe des verwendeten Metaklassensystems in MCS. Dieses erlaubte eine uniforme Sicht auf die verschiedensten Arten vorkommender Klassen: FLAVORS-Klassen als alte Implementierungsklassen, alte Frames als Wissensrepräsentationsobjekte, CLOS-Klassen als neue Implementierungsobjekte und schließlich neue Frames als neue Wissensrepräsentationsklassen. Dadurch konnten die neuen Entwicklungswerkzeuge schon während ihrer Implementierungsphase mit alten und neuen Wissensbasen demonstriert werden.

2.1.2 TINA

TINA ist ein objektorientierter Wissensrepräsentationsformalismus, der auf der *Konzeptsprache* KL-ONE beruht und in EuLisp als eine objektorientierte Programmiersprache integriert bzw. eingebettet ist [Kopp, 1996a]. Man kann TINA auch als Weiterentwicklung der Frames aus dem alten BABYLON und dem neuen babylon sehen. Für meine Arbeit ist TINA deshalb besonders interessant, weil seine Realisierung zu einem Zeitpunkt begann, als meine Implementierung von Τελος in einer Vorversion bereits verwendet werden konnte. Auf der anderen Seite konnte der Entwurf von Τελος noch beeinflußt werden, um die Anforderungen einer wiederverwendenden Implementierung von TINA weitgehend zu berücksichtigen.

2.1.3 KIKon

Im KIKon-Projekt wird ein Softwaresystem entwickelt, mit dem Kundenberater eines Telekommunikationsunternehmens, wie z.B. der Deutschen Telekom AG, komplexe Telekommunikationssysteme interaktiv konfigurieren können [Emde *et al.*, 1996]. Das System bietet an der Benutzeroberfläche einen graphischen Konfigurationseditor und einen komfortablen Tabellenauswahlmechanismus (FOCUS) an. Der ressourcenorientierte Konfigurator (ROK) prüft, ob die Benutzeraktionen zulässig sind und stellt die lokale und globale Konsistenz einer Konfiguration sicher. Das domänenspezifische Wissen ist im Produktkatalog und im abstrakten Modell der Produkte zusammengefaßt.

KIKonL

Um Telekommunikationsdienste und -komponenten zu beschreiben, wurde eine ressourcenorientierte Modellierungssprache KIKonL definiert. Diese erlaubt, von konkreten Produkten mit sehr vielen Merkmalen zu abstrahieren und nur konfigurationsrelevante Eigenschaften (Ressourcen) zu erfassen. Für Ressourcen und Komponenten können Generalisierungs- und Spezialisierungsbeziehungen ausgedrückt werden.

Zwei Komponenten (z.B. ein Telefon und eine Dose) sind dann miteinander verbindbar, wenn die eine Ressourcen bietet, die die andere benötigt. Aufgabe des Konfigurators ist es:

- die Konsistenz einer gegebenen Konfiguration von verbundenen Komponenten zu überprüfen und
- für gegebene Ressourcenforderungen automatisch eine Konfiguration zu finden, die diese erfüllt.

Die Implementierung von KIKonL muß also insbesondere dafür sorgen, daß die Anfrage des Konfigurators, ob zwei Komponenten verbindbar sind oder nicht, möglichst schnell beantwortet wird. Sowohl die Konsistenzprüfung als auch das automatische Konfigurieren beinhalten das Aufspannen großer Suchräume. Daraus folgt für das zugrundeliegende Objektsystem, daß die Instanzerzeugung möglichst schnell und speicherplatzsparend erfolgen muß.

Das KIKon-System wurde zunächst in Prolog und C++ implementiert. Danach wurde es in Java reimplementiert. Dabei hat sich gezeigt, daß das Objektsystem von JAVA nicht geeignet ist, die Anwendungsdomäne angemessen zu modellieren. Auf der anderen Seite sind die benötigten Mechanismen denen einer objektorientierten Programmiersprache sehr ähnlich. Um so bedauerlicher ist es, daß man in JAVA weder das Objektsystem spezialisieren noch die Syntax erweitern kann. In EULISP kann KIKONL als Erweiterung und Spezialisierung von TELOS realisiert werden.

Am Beispiel KIKONL wird deutlich, daß die für Lisp aufgestellten Anforderungen und erzielten Resultate auch auf andere, insbesondere funktionale oder imperative, Programmiersprachen unmittelbar übertragbar sind.

2.2 Anforderungen aus dem Softwarelebenszyklus komplexer Anwendungen

Komplexe Anwendungen werden üblicherweise von Teams über einen längeren Zeitraum entwickelt und gewartet. Alle Phasen im Softwarelebenszyklus solcher Anwendungen stellen ihre spezifischen, teilweise entgegengesetzten Anforderungen. Am bekanntesten ist einerseits das Wasserfallmodell [Boehm, 1981] und andererseits das Spiralenmodell [Boehm, 1988], worauf ich mich in dieser Arbeit beziehe.

Bei ihrer Betrachtung wird klar, daß die Verwendung von Sprachkonstrukten auch eng mit den zur Verfügung stehenden Entwicklungswerkzeugen wie Editor, Compiler, Interpreter, Debugger, Inspektor, Browser, Profiler, Projektmanager, etc. zusammenhängt. Dieser Zusammenhang wurde schon in [Sandewall, 1984], [Goldberg, 1984] etc. behandelt.

Doch zunächst stellt sich die Frage, was macht eigentlich die Komplexität von Softwaresystemen aus?

2.2.1 Komplexität von Softwaresystemen

Vielleicht haben die oben genannten Beispiele bereits eine intuitive Vorstellung hervorgerufen, worin die Komplexität von Softwaresystemen besteht. Systematisch und erschöpfend diese Frage zu beantworten, hieße das Problem zumindest theoretisch nahezu gelöst zu haben.

In erster Näherung kann man sagen, daß alle großen Systeme, die von vielen Beteiligten über lange Zeiträume entwickelt und genutzt werden, auch komplex sind. Darüberhinaus gehören explorative Softwaresysteme dazu, die neuartige Forschungshilfsmittel bereitstellen oder selbst Gegenstand der Forschung sind. Und schließlich sind hier interaktive Systeme zu nennen, deren Komplexität vor allem aus dem unvorhersehbaren Interaktionsablauf herrührt. Daraus ergeben sich drei Hauptgründe für die inhärente Komplexität von Softwaresystemen:

- die Probleme der Anwendungsdomäne sind komplex,
- die Interaktionsmöglichkeiten sind komplex,
- es ist schwierig, den Entwicklungsprozeß zu organisieren.

Wie auch in anderen wissenschaftlichen und technischen Disziplinen beruht ein grundlegender Lösungsansatz darauf, ein einfaches Architekturprinzip zu finden, das die Dekomposition des Gesamtproblems in lösbare Teilaufgaben erlaubt. Es gibt berechtigte Zweifel, ob dies für alle komplexen Probleme möglich ist.

Ein prinzipiell anderer Ansatz leitet sich aus dem Einganszitat dieser Arbeit ab: Bevor man damit beginnt, ein komplexes System zu realisieren, sollte man ein einfaches funktionierendes System konstruieren, das bereits die wesentlichen Eigenschaften des angestrebten komplexen Systems besitzt.[7] Natürlich kann es bei diesem Ansatz passieren, daß einige der wesentlichen Eigenschaften erst in der großen Version sichtbar werden. Dies trifft insbesondere für Performanzeigenschaften zu.

Hält man die heutigen Programmiersprachen und Entwicklungswerkzeuge dagegen, so ist man überrascht, wie wenig die o. g. Prinzipien befolgt werden. Gleichzeitig verwundert es nicht mehr, daß Softwareentwicklungsprojekte kaum kalkulierbar und ihre Ergebnisse alles andere als robust sind.[8]

Als unverzichtbare Mittel zur Komplexitätsreduktion in technischen und natürlichen Systemen sind Abstraktion, Klassifikation, Komposition, Modularisierung und Hierarchiebildung zu nennen [Booch, 1991].

2.2.2 Anforderungen aus Analyse und Design

Um die Analyse einer Anwendungsdomäne und das Design eines Softwaresystems zu unterstützen, werden spezielle problemnahe formale und semiformale Sprachen entwickelt. Dabei haben sich die objektorientierten Methoden einerseits und Entity-Relationship-Modelle andererseits als besonders nützlich erwiesen.

Die von einer Programmiersprache bereitgestellten Mittel können für die Analyse und das Design nicht direkt verwendet werden. Daher werden in der Regel Methoden und Entwicklungswerkzeuge eingesetzt, die mehr oder weniger auf eine Problemklasse und auf eine oder mehrere Programmiersprachen ausgerichtet sind. Bei dem späteren Schritt zur Implementierung kann es dann zu Brüchen kommen, wenn die Konzepte des Designs keine natürliche Entsprechung in der Implementierungssprache finden.

Um solche Brüche zu vermeiden, wurden in der Künstlichen Intelligenz operationale Wissensrepräsentationssprachen entwickelt, die es erlauben, die Domäne auf einem hinreichend abstrakten Niveau zu beschreiben. Gleichzeitig stellen sie geeignete Datenstrukturen bereit, um spezifische, auch alternative Problemlöser sowie Entwicklungswerkzeuge zu implementieren [Karbach, 1994]. Ihre Performanz und Integrationsfähigkeit in Standard-Softwareumgebungen stellen jedoch oft ein Problem dar.

Objektorientierte Programmiersprachen sollten daher Mittel bereitstellen, um effiziente und integrierte problemnahe operationale Sprachen als Erweiterung und Spezialisierung ausgewählter Sprachkonstrukte realisieren zu können. Dafür ist es notwendig, nicht nur zusätzliche Sprachkonstrukte realisieren zu können, sondern auch ungeeignete auszublenden.

[7] Die methodische Vorgehensweise, sogenannte Mikrosysteme zu bauen, habe ich bei Thomas Christaller auf der KIFS 1986 gelernt. Siehe auch [Schank und Riesbeck, 1981].

[8] Als kleines Beispiel möge man sich die Funktionsweise des Methodendispatches in Java klar machen oder gar die Handhabung dynamischer und lexikalischer Variablenbindungen in Common Lisp.

2.2.3 Anforderungen aus Implementierung, Evolution und Wartung

Während der Implementierung komplexer Software-Systeme wünschen sich Programmierer ein Höchstmaß an Flexibilität und Ausdrucksmächtigkeit der Programmiersprache und der Entwicklungswerkzeuge. Deshalb ist *rapid prototyping* so beliebt. Geht eine Anwendung in die Test- und Nutzungsphase, so rücken Robustheit und Effizienz der Ausführungsumgebung (Laufzeitsystem) in den Vordergrund. Später bekommen Portabilität, Wartbarkeit, Erweiterbarkeit und Anpaßbarkeit ein höheres Gewicht.

Wird ein System nur von einer Person entwickelt und gewartet, so spielen Überlegungen zur Modularisierung und Kapselung eine untergeordnete Rolle. Programmier- und Dokumentationsstil können dabei sehr individuell sein. Wird jedoch im größeren Team entwickelt, so wird es unvermeidlich, Namensräume zu trennen, unerwünschte Interaktionen auszuschließen, Schnittstellen zu vereinbaren, Änderungen systematisch und geordnet vorzunehmen, Versionsmanagement zu betreiben, Programmier- und Dokumentationsrichtlinien festzulegen und zu befolgen. Spätestens bei kommerziellen Systemen kommt auch das Qualitätsmanagement hinzu. Dabei müssen nicht unbedingt Zwangsjacken als Werkzeuge zum Einsatz kommen. Ist die Einsicht da, können einfache Konventionen sehr weit tragen. Aber leider muß man zu oft die Erfahrung machen,

> "daß man nicht mit der Einsicht der Leute rechnen darf." [Kopp, 1996b]

Um so wichtiger ist dann eine Programmiersprache, die eine einfache und transparente Semantik besitzt und bei aller Flexibilität für verständliche Programme sorgt.

Auch bei mittelgroßen Software-Entwicklungsprojekten wechseln die beteiligten Personen. Dabei müssen halbfertige Module übernommen, verstanden und weiterentwickelt werden. Selbst wenn es schneller ist, etwas neues *from scratch* zu programmieren, muß man so eine Entscheidung begründen und folglich auch das übernommene Programm zumindest im Prinzip verstanden haben. Wenn nun schon die (Programmier-) Sprache schwer verständlich ist, wie soll man dann die Botschaft in Abwesenheit des Programmierers verstehen?[9]

Aufgaben wie Integration von Teilsystemen, Portierung auf andere Soft- und Hardwareumgebungen verlangen ebenfalls, daß man Programme anderer versteht, auftretende Fehler ggf. nach Rücksprache beseitigen oder vermeiden kann.

In der Test- und Wartungsphase eines komplexen Systems kommt es darauf an, Fehler schnell zu erkennen und zu beseitigen, Anpassungen vorzunehmen, ohne dabei neue Fehler zu produzieren. Natürlich sind Programmfehler von Menschen gemachte Fehler und deshalb auch nicht nur technisch zu betrachten. Aber eine Programmiersprache ist das Kommunikationsmedium von Programmierern, sie muß daher als solches gesehen und auch entsprechend entworfen werden.

2.3 Designkriterien für objektorientierte Programmiersprachen

Nachdem nun der Verwendungskontext von objektorientierten Programmiersprachen anhand der Beispiele und anhand der verschiedenen Phasen des Softwarelebenszyklus skiz-

[9] Siehe hierzu [Christaller, 1997].

ziert wurde, können die Designkriterien plausibler und genauer formuliert werden.

Alle unten genannten Designkriterien sind so allgemein, daß sie inzwischen weitgehend konsensfähig und auf fast alle Sprachen und Systeme anwendbar sind [Kiczales *et al.*, 1991]. Grundlegende Prinzipien zur Beschreibung und Konstruktion technischer und natürlicher komplexer Systeme wie Abstraktion, Klassifikation, Komposition, Modularisierung und Hierarchiebildung [Booch, 1991] finden sich hier wieder. Generelle Designkriterien für Programmiersprachen sind aus der Perspektive des Stands der Technik z. B. in [Louden, 1994, S. 47 ff.] und [Fischer, 1994, S. 22 ff.] zu finden. Hier werden die Kriterien immer wieder aufgegriffen, um Design- und Implementierungsentscheidungen zu treffen.

Im folgenden geht es mir um eine erste spezifische Interpretation bezogen auf den Entwurf und die Implementierung effizienter Objektsysteme für komplexe Softwareanwendungen. Dabei wird deutlich, daß die Designkriterien selbst verschiedene, auch konkurrierende Facetten aufweisen, je nachdem für welche Phase des Softwarelebenszyklus und je nachdem für welche Tätigkeit ein Kriterium betrachtet wird. Beispielsweise kann man die Effizienz des Entwicklungsprozesses im Unterschied oder gar im Widerspruch zur Effizienz des Entwicklungsergebnisses sehen.

Andererseits hängen die Kriterien auch so miteinander zusammen, daß das eine nicht ohne das andere erfüllt werden kann. So sorgen angemessene Abstraktionen für eine einfachere Benutzung der Sprachkonstrukte, Modularisierung sichert schnellere Kompilation und somit kürzere Entwicklungszyklen usw.

2.3.1 Abstraktion

Abstraktion ist ein grundlegendes Mittel, wie Menschen mit Komplexität umgehen. Man reduziert die betrachteten Eigenschaften eines Gegenstandes auf das Wesentliche. Alles Unwesentliche wird zunächst vernachlässigt. Was wesentlich ist, legt der Betrachter spezifisch für den Zeitpunkt und den Zweck der Betrachtung fest. Im Kontext von Programmiersprachen ist man besonders an *sicherer Abstraktion* interessiert: alle Eigenschaften, die wir für das Abstraktum nachweisen, sollen auch für das Konkrete gelten.

Typisch für prozedurale Sprachen ist die algorithmische Abstraktion. Bei abstrakten Datentypen werden Datenstrukturen abstrahiert. In strikt objektorientierten Sprachen ist man immer gezwungen, beides gleichzeitig zu tun. Funktionale Sprachen abstrahieren generell von Zuständen bzw. Zustandsänderungen. Bestrebungen, von Implementierungsdetails zu abstrahieren, führen zur Separierung von Spezifikation und Implementierung oder gar zu strikt deklarativen Sprachen wie Logik[10], in denen man nur noch das Was und nicht das Wie ausdrückt. Sprachen mit automatischer Speicherverwaltung (engl. *garbage collection*) erlauben, von expliziter Speicheranforderung und Speicherfreigabe im Programm generell zu abstrahieren.

Abstrahiert man nur bis zu einem gewissen Grad von Entitäten, so ist es sinnvoll, die Zahl der Abstraktionsschritte nicht auf einen zu beschränken. Dann erhält man mehrere Abstraktionsebenen mit schrittweise abnehmender Granularität. Dieser Aspekt der Abstraktion ist für meinen Entwurf besonders wichtig, damit das Objektsystem so speziell

[10] Prolog ist zwar die wichtigste logische Programmiersprache, ist aber aus praktischen Überlegungen nicht strikt deklarativ.

wie nötig und so allgemein wie möglich wiederverwendet werden kann. Die Spracherweiterungsprotokolle müssen für die verschiedenen Spezialisierungsaufgaben angemessene Granularitätsstufen anbieten.

2.3.2 Klassifikation

Klassifikation ist ein weiteres grundlegendes Mittel, um mit Komplexität umzugehen. Dabei werden Objekte mit ähnlichen Eigenschaften in entsprechende Klassen eingeteilt. Die Klassen selbst können anschließend in Generalisierungs- und Spezialisierungs- oder auch Aggregationsbeziehung gesetzt werden, was zu entsprechenden Klassenhierarchien führt. Eine gute Klassifikation ist eine wichtige Voraussetzung für jede gute Theorie. [11]

Eine systematische Wiederverwendung von Softwarebausteinen kann erst dann erfolgen, wenn man sie für die jeweiligen Aufgaben geeignet klassifiziert und wenn eine Klassifikation eine hinreichend breite Akzeptanz findet. Hier liegt aber auch das Hauptproblem, warum es in der Softwareproduktion so schwierig ist, zu einem hohen Wiederverwendungsgrad zu kommen. Das Klassifizieren ist inhärent subjektiv und zweck-orientiert. Die eine allgemeingültige, objektive Klassifikation gibt es nicht. Und auch für genau ein Individuum gibt es keine endgültige, perfekte Klassifikation. So lange sich Gegenstandsbereiche ändern und die Erfahrungen der Menschen damit, müssen auch ihre Klassifikationen ständig angepaßt werden.

Im Kontext der objektorientierten Programmierung kann man zwei grundsätzlich verschiedene Fälle unterscheiden. Im einfachen Fall sind die Klassifikationskriterien statischer Natur, d. h. die interessierenden Klassen sind zum Zeitpunkt der Programmspezifikation bekannt und können unmittelbar als solche im Programm definiert werden. Im schwierigeren Fall sind die Klassifikationskriterien dynamischer Natur, d. h. die interessierenden Klassen sind zum Zeitpunkt der Programmspezifikation noch nicht bekannt. Sie können erst zur Ausführungszeit eines Programms berechnet werden. Will man das Klassifizieren umfassend unterstützen, so müssen auch entsprechende Sprachkonzepte und syntaktische Konstrukte sowohl für statisches als auch für dynamisches Klassifizieren angeboten werden.

2.3.3 Spezialisierung und Generalisierung

Als übergeordnetes Prinzip, wie man Softwarebausteine klassifiziert, betrachte ich das *Generalisieren und Spezialisieren* von Objektklassen. Ihre Unterstützung ist daher die wichtigste Forderung für meinen Entwurf. Legt man sich in diesem Punkt nicht einmal auf so einem abstrakten Niveau fest, wie bei C++ zum Beispiel, so wird schnell unklar, was man eigentlich unterstützen will. Versuche, zu viel zu unterstützen, führen unweigerlich zur Überfrachtung, zu Mißbrauch, Mißverständlichkeit und letztlich zu einer geringeren Wiederverwendbarkeit entsprechender Softwarebausteine. Erfahrungen von Praktikern und Untersuchungen zu objektorientierten Softwaremetriken bestätigen diese Einschätzung [Lorenz, 1995, S.60].

[11] Nach neueren Erkenntnissen aus der Hirn- und Kognitionsforschung können Menschen ihre Umgebung nur kategorisierend und klassifizierend wahrzunehmen [Poeppel, 1985].

Die Unterstützung der Klassifikation sowie des Spezialisierens und Generalisierens ist das auszeichnende Merkmal objektorientierter Programmiersprachen. Sie bildet einen Schwerpunkt dieser Arbeit.

2.3.4 Komposition

Die Komposition ist als Modellierungsprinzip deutlich von der Klassifikation zu unterscheiden. Komposition beschreibt die Teil/Ganzes- bzw. die Aggregationsbeziehung, während die Klassifikation Ähnlichkeitsbeziehungen ausdrückt. Die Mißachtung dieses Unterschieds führt zu verwirrenden Konstruktionen, wenn beispielsweise Vererbung als ein syntaktisches Konstrukt einer Programmiersprache für beide Arten von Beziehungen verwendet wird.

Eine objektorientierte Programmiersprache sollte die Komposition von Struktur und Verhalten von Objekten unterstützen, indem komplexe Objekte aus einfacheren zusammengesetzt werden können. Dabei muß aber der oben genannte Unterschied beachtet werden.

Komposition als ein allgemeines Kriterium für Programmiersprachen verlangt, daß Ausdrücke flexibel zusammensetzbar sind. Beispielsweise sollten an Argumentstellen bei Funktions- oder Prozeduraufrufen nicht nur Variablen, sondern beliebige auszuwertende Ausdrücke stehen dürfen. Dies ist in Java nicht gegeben, da man z. B. Array-Literale eben nicht an Argumentstelle verwenden darf.

2.3.5 Hierarchiebildung

Nach Booch [1991] spielen in komplexen Systemen zwei Hierarchiearten eine wichtige Rolle: die Klassenhierarchie und die Objekthierarchie. Wie schon oben erwähnt, wird die erste aus den Ähnlichkeitsbeziehungen abgeleitet, während die zweite sich aus der Teil/Ganzes-Beziehung ergibt.

Im Mittelpunkt meiner Betrachtung werden Klassenhierarchien stehen. Dabei werden insbesondere einfache Vererbung, multiple Vererbung und die Mixin-Vererbung behandelt. Natürlich können auch Module hierarchisch aufgebaut werden. Man spricht dann auch von geschachtelten Modulen (siehe nächstes Kriterium).

Im Kontex meines Sprachdesigns wird die Hierarchiebildung von den Kriterien Klassifikation und Komposition subsumiert, siehe Abbildung 2.2.

2.3.6 Modularisierung

Auch wenn einige Programmiersprachen keine Unterscheidung zwischen Klassen und Modulen vornehmen, so ist es doch offensichtlich, daß Klassifikation und Modularisierung unterschiedliche Modellierungskonzepte sind [Szyperski, 1992]. Eine Klassifikation von Objekten liefert zwar eine mögliche Modularisierung; und auch Module kann man klassifizieren. In der Regel dienen sie jedoch einem unterschiedlichen Zweck. Nach Booch [1991] drücken Klassen die logische Struktur eines Systems aus, während Module die physische Architektur darstellen. Als Beispiel nennt er elektronische Bauteile: es werden nicht alle NAND- bzw. NOR-Gatter zu jeweils einem Modul (Baustein) zusammengefaßt, sondern beide Gatterarten werden in verschiedenen Schaltkreisen zusammengeschaltet.

In Programmiersprachen dienen Module der Schnittstellenbildung; sie bilden die mittelgroßen Übersetzungseinheiten, spezifizieren Import/Export-Beziehungen, unterstützen das *information hiding* und „gehen Hand in Hand mit der Datenkapselung“ [Booch, 1991]. Auf der technischen Seite der Sprachkonstrukte regeln Module die Sichtbarkeits- und Gültigkeitsbereiche von Konstanten- und Variablenbindungen. Auf der konzeptuellen Seite realisieren sie einen vom Programmierer eingegrenzten Funktionskomplex.

So wurde für EuLisp die Designentscheidung getroffen, daß das Modul- und das Objektsystem orthogonale Sprachkonzepte darstellen. Das Kriterium der Modularisierung bedeutet hier sogar, daß das Objektsystem modular aufgebaut werden soll. Dadurch wird u. a. die effiziente Übersetzung von Anwendungen und die Erzeugung effizienter schlüsselfertiger Applikationen unterstützt. Da die Benutzung einiger Sprachkonstrukte, insbesondere der Spracherweiterungsprotokolle, seinen Preis hat, sollte nur derjenige ihn bezahlen müssen, der die Leistung auch nutzt. Die Nutzungsabsicht wird in Modulen durch entsprechende Importangaben spezifiziert.

2.3.7 Reflektion

Unter der Reflektion in Programmiersprachen [Smith, 1984] versteht man die Fähigkeit, Elemente ihrer Realisierung selbst zum Gegenstand der Betrachtung zu machen. Im einfachsten Fall bedeutet dies, daß man z. B. in einer funktionalen Sprache eine Funktion an Argumentstelle eines Funktionsaufrufs verwenden kann. Hier spricht man auch von Funktionen höherer Ordnung. Am anderen Ende des Reflektionsspektrums steht die Möglichkeit, die Ausführung des Programms selbst zu beeinflussen. Letzteres bringt natürlich einige semantische Schwierigkeiten mit sich, während das erste in Lisp und anderen funktionalen Programmiersprachen so selbstverständlich ist, daß kaum jemand es als Reflektion wahrnimmt.

Reflektion liegt in Programmiersprachen dann vor, wenn man Programme über Programme schreiben möchte, wie z. B. Compiler oder Debugger. An diesen Beispielen sieht man auch die unterschiedlichen Arten der Reflektion in Programmiersprachen. Ein Compiler reflektiert zur Übersetzungszeit über den Quellcode eines Programms. Ein Debugger reflektiert zur Ausführungszeit über das dynamische Verhalten eines Programms. Meistens wird nur letzteres als Reflektion bezeichnet.

Für Objektsysteme heißt das, daß man beispielsweise über Klassen als Objekte reflektieren können muß, wenn man neue Arten von Klassen mit erweiterter Funktionalität realisieren möchte ohne schon vorhandene Funktionalität nochmal selbst implementieren zu müssen. Die oben genannten Beispiele komplexer Softwaresysteme benötigen genau dieses.

Reflektive Sprachmittel sollten so entworfen werden, daß sie die Robustheit und Effizienz des Objektsystems insgesamt nicht beeinträchtigen. Hierin liegt ein Schwerpunkt meiner Arbeit.

2.3.8 Erweiterbarkeit und Wiederverwendbarkeit

Die Forderung nach Erweiterbarkeit ist vor allem für komplexe Softwaresysteme unverzichtbar. Glaubt man Statistiken, so machen Erweiterungen von vorhandenen eingesetzten Systemen 80 % der gesamten Softwareproduktion aus, während Neuimplementierungen

nur 20 % ausmachen [BMFT, 1994, S.12]. Grund genug sich um die Erweiterbarkeit zu kümmern!

Auch wenn die Grundideen zur Objektorientierung in der Informatik schon etwa 30 Jahre alt sind und in den letzten 10 Jahren sich durchaus etabliert haben, gibt es immer noch keine allgemein akzeptierte Theorie und Praxis objektorientierter Konzepte und Sprachkonstrukte. Und selbst wenn es sie gäbe, sind die spezifischen Anforderungen verschiedener Anwendungsdomänen so unterschiedlich, daß eine Programmiersprache nicht jeden Aspekt unterstützen kann. Wichtiger als eine vollständige Abdeckung anzustreben, sind deshalb erweiterbare Konzepte und wiederverwendbare Realisierungen. Dann kann ein Systemprogrammierer die Systemimplentierungssprache an die Anwendungsprobleme anpassen, so daß Anwendungsprogrammierer ihre Designmodelle in der Implementierung wiederfinden können.

Voraussetzung für die Erweiterbarkeit und Wiederverwendbarkeit von Objektsystemen sind offene Erweiterungsprotokolle. Dies steht zunächst im Widerspruch zu Forderungen nach maximaler Effizienz und Robustheit, denn jede Offenlegung einer Implementierung von Sprachkonstrukten erschwert interne Optimierungen und erleichtert unerwünschte Effekte von außen. Auf der anderen Seite ist die Erweiterbarkeit ein gutartiges Mittel auf dem Weg zur Anpaßbarkeit im Unterschied zur Modifizierbarkeit, die in CLOS den Vorrang erhielt.

2.3.9 Einfachheit

Schon bei der Charakterisierung komplexer Softwaresysteme wurde deutlich, daß man Komplexität nur mit einfachen Mitteln in den Griff bekommt. Ob etwas einfach ist oder kompliziert, kann man leider nicht so eindeutig beantworten. Und für wen soll etwas einfach sein? Für den Benutzer oder für den Anbieter? Welche Werkzeuge darf man für die Benutzung voraussetzen, und welche Hilfsprimitive für den Anbieter? Ob z. B. die Syntax einer Programmiersprache als einfach empfunden wird, hängt weitgehend von Gewohnheiten und vom benutzten Editor ab.

Zunächst meine ich mit Einfachheit, daß die Konzepte leicht verständlich sind. Eine einfache formale Semantik garantiert nocht nicht, daß der Sachverhalt auch leicht zu verstehen und effizient zu implementieren ist. Daher müssen die Konzepte auf verständliche Weise auch informell beschrieben werden können. Ebenso muß der Implementierungsaufwand sowie die algorithmische Komplexität des Ergebnisses abschätzbar sein.

Die Semantik von Sprachkonstrukten in Programmfragmenten sollte lokal und ohne Testläufe erschließbar sein. Für eine gegebene Definition sollte z. B. auch sicher sein, daß sie an keiner anderen Stelle im Programm zunichte gemacht werden kann, wie dies in CommonLisp Anwendungen häufig der Fall ist. Insofern bedeutet einfache Benutzung nicht, daß man "mal eben schnell" alles Beliebige redefinieren kann, ohne über Zusammenhänge nachgedacht zu haben. Einfache Benutzung ist das Gegenteil von *quick and dirty*, sie darf nicht auf Kosten der Robustheit erreicht werden.

Mit der Einfachheit ist auch die Forderung nach *Uniformität* verbunden. Ähnliche Aufgaben sollten auf gleiche Weise gelöst werden. Ausnahmen und Sonderfälle sollten vermieden werden. Dies beginnt mit der Syntax und endet mit Erweiterungsprotokollen, die es er-

lauben, Objekte der Anwendungsdomäne auf gleiche Weise zu behandeln, wie die Objekte der Programmiersprache.

Es muß sorgfältig unterschieden werden, welche Flexibilität von der Programmentwicklungsumgebung zu erbringen ist, und welche Manipulationsmöglichkeiten zur Laufzeit eines Programms erforderlich sind. Und diese Differenzierung muß für den Programmierer transparent sein.

2.3.10 Orthogonalität

Mit der Einfachheit ist eng die Forderung nach *Orthogonalität* der Sprachkonstrukte verbunden. Damit ist gemeint, daß die einzelnen Konstrukte nicht mit Aufgaben überfrachtet werden sollten, die von anderen Konstrukten schon erledigt werden können. Natürlich muß es nicht unbedingt nur einen Weg geben, etwas zu tun. Man sollte aber das Rad nicht immer wieder neu erfinden. Bewährte Konzepte der strukturierten Programmierung, wie funktionale Komposition, explizite Schnittstellenbildung zwischen Programmteilen, z. B. mit Modulen, sollten nicht mit aller Gewalt in das objektorientierte Paradigma hinein interpretiert werden. Das Kriterium der Orthogonalität bildet ein Gegengewicht gegenüber dem Hang zur Übergeneralisierung in der Informatik. Möglichst allgemeine Lösungen sind nach meiner Auffassung nicht immer nützlich. Zum Beispiel dann nicht, wenn dadurch einfache Probleme kompliziert gelöst werden müssen. Darunter leidet meistens die Effizienz.

Will man orthogonale Sprachkonstrukte entwerfen, so muß nicht nur sorgfältig generalisiert, sondern auch pragmatisch differenziert werden.

2.3.11 Robustheit

Unter *Robustheit* eines Systems versteht man im allgemeinen seine Fähigkeit, auf unvorhersehbare Eingbaben bzw. Störungen gutartig zu reagieren und dabei noch brauchbare Resultate zu liefern. Im Kontext komplexer Softwaresysteme beinhaltet Robustheit auf jeden Fall *Korrektheit* , geht aber darüber hinaus. Für objektorientierte Sprachkonstrukte bedeutet Robustheit insbesondere, daß Klassen, die spezialisiert werden, sich unverändert verhalten. Mehrere Subklassen einer Klassen dürfen sich gegenseitig nicht stören. Die Integrität der Superklassen und der Nachbarklassen muß beim Spezialisieren gewährleistet sein.

Robustheit ist für komplexe Anwendungen noch wichtiger als Effizienz. Robuste Programme kann man leichter optimieren, als man schnelle Programme robust machen kann. Eine mäßige Performanz kann von Anwendern noch hingenommen werden, nicht aber unerklärliches Verhalten oder gar Programmabstürze.

Um Robustheit zu erreichen, muß das zugesicherte Verhalten von Programmbausteinen auch garantiert werden können. Hier ist die Stelle, an der Software-Verifikation beginnt. Formale Software-Verifikation reicht aber alleine nicht aus. Sie ist oft auch nicht mit angemessenem Aufwand vollständig zu erreichen. Andere Techniken der Qualitätssicherung, insbesondere bei Realzeitsystemen und kritischen Anwendungen, müssen hinzukommen. Will man Anpaßbarkeit unterstützen, so muß dies dann über den Weg von Erweiterun-

gen geschehen. Erweiternde Klassen dürfen das Verhalten der erweiterten Klasse, von der andere Programmteile abhängen können, nicht verändern.

In CommonLisp wird die Forderung nach Robustheit nur unzureichend berücksichtigt. Methoden können für beliebeige Klassen hinzukommen und andere ersetzen. Klassen können redefiniert werden, und ob die automatischen Anpassungen abhängiger Klassen ausreichen ist nicht gesichert.

Erweiterungsprotokolle sollte man möglichst funktional spezifizieren, um eine höhere Robustheit der Erweiterungen zu erreichen. Das bedeutet, daß Eingangsgrößen für das Objektverhalten als Parameter von Methoden spezifiziert und bei Bedarf auch über eine Kette von Methodenaufrufen durchgereicht werden sollten. Dadurch kann man vermeiden, Zwischenzustände zu spezifizieren und unnötige Sequentialisierungen festzulegen.

Konzepte wie Redefinieren in CLOS müssen sorgfältig überdacht werden, um nicht in Konflikt mit der Forderung nach Robustheit zu geraten. Auf Sprachkonstrukte, die mit der Robustheit unvereinbar sind, muß spätestens bei Programmauslieferung verzichtet werden.

Die Forderung nach Einfachheit trägt auch direkt zur höheren Robustheit bei.

2.3.12 Effizienz

Für viele komplexe Anwendungen ist ein gewisses Mindestmaß an Effizienz eine notwendige Voraussetzung, um überhaupt einsetzbar zu sein. Meistens steckt in komplexen Anwendungen viel Optimierungspotential. Insbesondere dann, wenn nach der Methode *rapid prototyping* entwickelt wurde, ohne die Bereitschaft, Module, die im Prinzip funktionieren, nochmal zu überarbeiten. Unabdingbare Werkzeuge für Performanzanalysen komplexer Softwaresysteme sind sogenannte Profiler, die genaue Daten liefern, in welchen Programmteilen wieviel Zeit verbraucht wird.

Ist das Objektsystem der verwendeten Programmiersprache nicht effizient genug, pflanzt sich das Problem fort, und es bleibt nur der Ausweg, auf objektorientierte Sprachmittel zu verzichten oder die Programmiersprache ganz zu wechseln. So wurden viele KI-Werkzeugsysteme von CommonLisp nach C++ portiert, weil es Performanz- und Integrationsprobleme gab.

Wie schon oben erwähnt, steht die Erweiterbarkeit zunächst im Widerspruch zur Effizienz, weil ein optimierender Compiler weniger Annahmen über den Verwendungskontext machen darf, wenn z. B. Methoden auch auf Instanzen potentieller Subklassen angewandt werden können. Ebenso negativ wirkt die Forderung nach Robustheit, weil ggf. mehr Typprüfungen zur Laufzeit durchgeführt werden müssen.

Um eine höhere Effizienz zu erreichen, muß man zunächst zwischen der Übersetzungs-, Lade- und Ausführungszeit einer Anwendung unterscheiden. Längere Ladezeiten sind weniger kritisch als schlechte Ausführungszeiten. Das Wissen darüber kann man im Entwurf und in der Implementierung ausnutzen, um ein angemessenes Gesamtverhalten zu erreichen.

Der Entwicklungsprozeß wird durch die Erweiterbarkeit und Wiederverwendbarkeit des Objektsystems deutlich beschleunigt. Die reflektiven Sprachanteile mit den offenen Erweiterungsprotokollen werden vor allem für große und komplexe Anwendungen konzipiert.

Daher kommen auch nur für solche ihre Stärken voll zum Tragen. Werden sie unbedacht verwendet, können ihre Kosten den tatsächlichen Nutzen übersteigen.

2.3.13 Verursacherprinzip

Im Zusammenhang mir der Forderung nach Effizienz soll das Verursacherprinzip gelten, d. h. daß nur die Programmteile, die kostenträchtige Sprachkonstrukte verwenden, entsprechende Performanznachteile in Kauf nehmen müssen. Andere Teile, die z. B. ohne Redefinieren von Klassen auskommen oder auf reflektive Sprachmittel ganz verzichten, sollten auch so optimierbar sein, wie in statisch analysierbaren Sprachen. Hier kann das Modularisierungsprinzip auf den eigenen Entwurf angewendet werden, um sparsame von kostenträchtigen Sprachmitteln zu separieren und um dem Benutzer transparent zu machen, was er benutzt und mit welchen Konsequenzen.

2.3.14 Zusammenfassung der Kriterien

In der Abbildung 2.2 fasse ich die Diskussion der Designkriterien graphisch zusammen. Die Pfeile zeigen, wie die Kriterien andere verstärken. Kanten ohne Pfeile drücken ein gewisses Spannungsverhältnis aus. Natürlich gibt es im Prinzip zwischen allen Kriterien Wechselwirkungen. Hier werden nur die wichtigsten Beziehungen für mein Sprachdesign ausgedrückt.

Dabei wird deutlich, daß der Entwurf und die Implementierung effizienter Objektsysteme eine komplexe Optimierungsaufgabe darstellen, die kaum ein für allemal optimal gelöst werden kann. Neue Anforderungen oder auch Hilfsmittel können zu neuen Lösungen führen. Diese Arbeit zeigt aber, wie die teilweise entgegengesetzten Ziele in Einklang gebracht werden können und so, gemessen an den heute verwendeten objektorientierten Sprachen, ein wichtiger Schritt nach vorn gemacht werden kann.

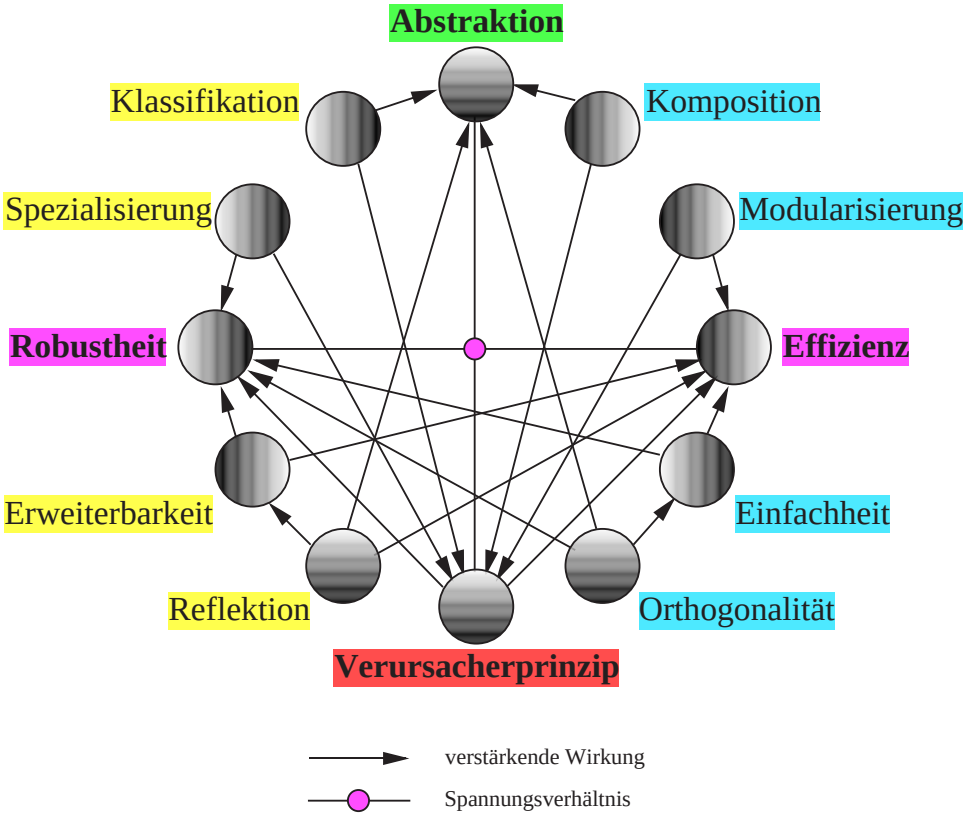


Abbildung 2.2: Wechselwirkungen zwischen den Kriterien.

Kapitel 3

Stand der Technik

> *Language design is one of the most difficult and poorly understood areas of computer science.*
>
> Louden, Kenneth C. 1993. Programming Languages - Principles and Practice, PWS-KENT, p. 47.

> *The names* derived *and* base *were chosen because I never could remember what was* sub *and what was* super *...*
>
> Stroustrup, Bjarne 1994. The Design and Evolution of C++, Addison-Wesley, S. 49.

In diesem Kapitel gebe ich einen Überblick über den Stand der Technik bei objektorientierten Programmiersprachen. Dabei werden zunächst die grundlegenden Sprachkategorien gegeneinander abgegrenzt, die auf unterschiedlichen Verarbeitungsmodellen beruhen und einen entsprechenden Programmierstil nach sich ziehen. Allerdings führen diese Kategorien nicht zu einer disjunkten Klassifikation der Programmiersprachen. Anschließend werden die gängigen Begriffe und Konzepte aus der Objektorientierung erläutert. Im letzten Abschnitt werden exemplarisch einige der wichtigsten und für diese Arbeit relevanten objektorientierten Programmiersprachen vorgestellt. Dabei gehe ich auch auf die jeweilige historische Entwicklung und die getroffenen Designentscheidungen ein.

3.1 Programmiersprachen, Verarbeitungsmodelle und Programmierstil

Das auszeichnende Merkmal universeller Rechenmaschinen, im Unterschied zu speziellen, ist die Möglichkeit, sie beliebig zu programmieren. Was und wie eine Maschine berechnen soll, wird ihr in Form einer Befehlssequenz mitgeteilt. Der Befehlssatz, die Sprachkonstrukte, sind unmittelbar durch die Konstruktion der Maschine vorgegeben und legen die Maschinensprache fest. So wurden die ersten digitalen Computer unmittelbar in der jeweiligen Maschinensprache programmiert. Ein Programm war demnach eine Folge von Bitmustern – eine Repräsentation, die nicht zu den Stärken menschlicher kognitiver Fähigkeiten zählt. So wurden im nächsten Schritt Befehlen und Speicheradressen “sprechende”

Symbole zugeordnet[1], so daß Programme in Assemblersprache notiert werden konnten. Vor der Ausführung eines Programms durch eine Maschine, mußte es nun erst in Maschinensprache übersetzt werden. Da nun die physikalische Realisierung der Maschine in den Hintergrund trat, rückte der Gegenstand einer Berechnung, das zu lösende Problem, in den Vordergrund. Es lag also nahe, von maschinennahen zu aufgabenadäquaten Programmiersprachen zu abstrahieren, so daß es einfacher wurde, Programme zu schreiben, zu verstehen und mit Menschen zu kommunizieren. Mit fortschreitender Abstraktion der Sprachkonstrukte wurde der Übersetzungsschritt in Maschinensprache immer aufwendiger und komplexer. Mit dem Design von höheren Programmiersprachen ist also auch unmittelbar der Compilerbau verbunden.

Höhere Programmiersprachen [Horowitz, 1983] beginnend mit Fortran, Lisp und Cobol haben inzwischen eine vierzigjährige Geschichte hinter sich. Etwa zehn bis zwanzig Jahre jünger sind die ersten objektorientierten Programmiersprachen wie Simula [Dahl und Nygaard, 1966] und Smalltalk [Goldberg und Robson, 1983]. Zur gleichen Zeit wurden auch objektorientierte Wissensrepräsentationssprachen wie semantische Netze [Quillian, 1967], Frames [Minsky, 1975], ACTOR [Hewitt *et al.*, 1973] u. a. entwickelt, die auch objektorientierte Programmiersprachen beeinflußten. Zur Geschichte der Programmiersprachen enthält [Bergin und Gibson, 1996] sehr aufschlußreiche Beiträge der beteiligten Sprachdesigner.

Zusammenfassend kann man sagen, daß für die Entwicklung höherer Programmiersprachen zwei übergeordnete Ziele maßgeblich sind: erstens soll der Software-Entwicklungsprozeß anwendungsorientiert unterstützt werden und zweitens muß das Entwicklungsergebnis auf einem physikalischen Rechner effizient ausführbar sein. Es ist offensichtlich, daß diese Ziele miteinander konkurrieren. Welche Gewichtung man nun für eine konkrete Sprache vornimmt und in welcher Reihenfolge man versucht, die Ziele zu erreichen, kann durchaus unterschiedlich sein.

3.1.1 Abstraktion in Programmiersprachen

Nach Louden [1994] können Abstraktionen in Programmiersprachen in zwei Kategorien eingeteilt werden: *Datenabstraktion* und *algorithmische Abstraktion*. Je nach Abstraktionsgrad können verschiedene Abstraktionsebenen unterschieden werden. Ob die Abstraktionsebenen fest vorgegeben sind oder vom Programmierer selbst definiert werden können, macht im wesentlichen die Flexibilität und Ausdrucksmächtigkeit einer Programmiersprache aus.

Datenabstraktion abstrahiert von konkreten Eigenschaften der Daten und ihrer internen Repräsentation. Speicherzellen werden mit Namen assoziiert und *Variablen* genannt. Unterschiedliche Datenarten werden zu *Datentypen* abstrahiert, Variablen können dann auch typisiert werden. Konsequente Datenabstraktion führt zum Konzept *abstrakter Datentypen* [Liskov, 1974], [Bishop, 1986]. Typisch hierfür sind Module in Modula-2 [Wirth, 1985a] oder Packages in Ada [Nagl, 1982].[2] Algorithmische Abstraktion abstrahiert den Kontrollfluß im Programm, wie zum Beispiel Zuweisungen, Schleifen, Fallunterscheidungen oder

[1] Was letztlich auch die *symbol hypothesis* ausmacht [Newell und Simon, 1972].

[2] Dabei fällt auf, daß Abstraktion von Datentypen hier sehr eng, auf Sichtbarkeit beschränkt, verstanden wird, die Datentypen als solche bleiben wie sie sind, nämlich konkret. Oder anders ausgedrückt, das einzige Mittel zur Datenabstraktion ist die Programm-Modularisierung.

Prozeduren. Schon die *goto*-Anweisung stellt eine, wenn auch keine hinreichende, Abstraktion von konkreten Sprunganweisungen einer Maschine dar. Prozeduren und Funktionen erlauben, Berechnungen auf einer beliebigen Abstraktionsebene zu beschreiben. Abstrahiert man Aspekte der Nebenläufigkeit und Parallelität, so erhält man Konzepte wie Coroutinen in Modula-2 oder Prozesse in Ada.

In strikt objektorientierten Programmiersprachen ist Datenabstraktion und algorithmische Abstraktion eng gekoppelt. Man kann nicht das eine ohne das andere tun.

3.1.2 Verarbeitungsmodelle und Programmierstil

Bemerkenswerterweise verschwindet die Vorstellung darüber, wie die Maschine Konstrukte höherer Programmiersprachen ausführt keineswegs. Sie wird zum kognitiven Verarbeitungsmodell [Christaller, 1986], das sich nicht nur von der konkreten Maschine, sondern auch von abstrakten, formal spezifizierten Maschinenmodellen [Stoyan und Görz, 1984] unterscheidet. Letztere werden im Compilerbau auch als *virtuelle Maschine* bezeichnet und erhöhen die Portabilität von Programmiersprachen [Goldberg und Robson, 1983], [Gosling *et al.*, 1996]. Nimmt man als Verarbeitungsmodell zum Beispiel die von Neumann-Rechnerarchitektur an, so werden bestimmte Abstraktionsarten in einer Programmiersprache begünstigt, nämlich die der imperativen oder prozeduralen Sprachen. Und selbst, wenn eine Programmiersprache mehrere Verarbeitungsmodelle zuläßt, was im Prinzip für alle universellen (Turing-äquivalenten) Programmiersprachen zutrifft, führt das kognitive Verarbeitungsmodell im Kopf der Programmierer zu einem bestimmten Programmierstil.

Insofern ist es mehr der Programmierstil, den man als imperativ oder funktional klassifizieren kann, als die Sprache. Dennoch unterscheiden sich die Sprachen sehr deutlich darin, welchen Programmierstil sie besser unterstützen. In diesem Sinne ist die folgende, allgemein etablierte, Klassifikation in

- imperative,
- funktionale,
- deklarative und
- objektorientierte

Sprachen zu verstehen. Andere Sprachaspekte führen zur Bildung weiterer Kategorien, wie

- regel-orientierte,
- constraint-orientierte,
- parallele,
- echtzeit-fähige

Sprachen etc., die hier nicht weiter ausgeführt werden.

Der *imperative Programmierstil* zeichnet sich dadurch aus, daß globale Speichervariablen deklariert werden, deren Werte durch Zuweisungen geändert werden, die in sequentiellen Befehlsfolgen ausgeführt werden. Berechnungsergebnisse entstehen als Seiteneffekte in Speicherzuständen. Dem ist als Verarbeitungsmodell unmittelbar die von Neumann-Architektur zugrundegelegt bzw. die Turingmaschine. Typische imperative Sprachen sind Fortran, Pascal, C, Ada, Cobol und Basic.

Nachteil der vorgeschriebenen sequentiellen Anweisungsspezifikation bei imperativen Programmiersprachen ist die dadurch bedingte Überspezifikation eines Lösungswegs. Oft sind Teile komplexer Berechnungen voneinander unabhängig. Da sie aber in einer Reihenfolge hingeschrieben werden müssen, kann die Möglichkeit ihrer nebenläufigen Ausführung nicht ausgedrückt und vom Compiler im allgemeinen auch nicht abgeleitet werden. Obwohl generell Überspezifikationen auch das Verstehen von Programmen erschweren, weiß man jedoch aus der Praxis, daß es Programmierern (und Menschen überhaupt) wesentlich leichter fällt, sequentiell als parallel zu denken[3].

Der *funktionale Programmierstil* gründet auf der mathematischen Theorie rekursiver Funktionen [Hudak, 1989], [Hughes, 1990]. Berechnungen werden in Form von Funktionsdefinitionen und Funktionsanwendungen beschrieben. Funktionale Sprachen werden daher auch als applikative Sprachen bezeichnet. Komplexe Funktionen werden aus einfachen durch Kombination, Fallunterscheidung und Rekursion zusammengesetzt. Die Semantik von funktionalen Sprachen kann ohne Transformationen von Speicherzuständen definiert werden, sondern durch Vorschriften, wie Ausdrücke in eine Normalform reduziert werden können. Man spricht daher auch von Reduktionsmaschinen [Hommes *et al.*, 1980], [Zimmer, 1991]. Auf Speichervariablen und Zuweisungsanweisungen kann ganz verzichtet werden. Stattdessen gibt es formale Parameter von Funktionen, die bei Funktionsanwendung an entsprechende Werte konkreter Argumente gebunden werden. Als Argumente können wiederum zusammengesetzte Ausdrücke verwendet werden. Anstelle von Schleifen genügt das Mittel der Rekursion, das wesentlich stärker die algorithmische Abstraktion unterstützt als Schleifen.

Das Verarbeitungsmodell bei funktionalen Sprachen unterscheidet sich also deutlich von dem bei imperativen Sprachen. Es entspricht weitgehend der mathematischen Sicht von Funktionen. Dies erleichtert, beweisbare Eigenschaften von Programmen zu erhalten [Goerigk, 1993]. Da funktionale Programme implizit parallele Berechnungen zulassen, kann dies auf entsprechenden Rechnern auch ausgenutzt werden. Typische funktionale Sprachen sind Lisp, Scheme, ML bzw. Standard ML [Milner *et al.*, 1990], [Paulson, 1991] Miranda [Turner, 1990], Haskell [Hudak und Wadler, 1992]. Da die zwei letzten gänzlich auf imperative Sprachkonstrukte verzichten, werden sie auch strikt funktional genannt.

Da das Verarbeitungsmodell funktionaler Sprachen nicht der von Neumann-Architektur entspricht, gab es zunächst das Problem, sie effizient für verfügbare Rechner bereitzustellen. Dieses Problem ist inzwischen beseitigt [Bretthauer *et al.*, 1994], [Kluge, 1997], wenn auch das Vorurteil sich immer noch hält.

Der *deklarative Programmierstil* leitet sich aus der mathematischen Logik und insbesondere aus der Aussagenlogik und der Prädikatenlogik erster Stufe ab. Er wird daher auch der *logische Programmierstil* genannt. Ein logisches Programm drückt aus, was berechnet

[3]Der Grund liegt darin, daß Denken internes Sprechen ist und sprechen fordert eine Sequenzialisierung der Vorstellungen (Imagination), die selbst eher assoziativ, distribuiert und parallel sind [Poeppel, 1985].

werden soll, nicht wie es berechnet werden soll. Es besteht aus einer Menge von logischen Ausdrücken (Formeln). Das primäre Ergebnis seiner Ausführung bzw. seines Beweises ist die Mitteilung, ob die Formeln wahr oder falsch sind. Da man als Programmierer aber nicht nur wissen möchte, ob eine Lösung existiert, sondern auch wie sie aussieht, benutzt man logische Variablen, die eine oder mehrere Lösungen repräsentieren und als Seiteneffekt der Beweisführung an Werte gebunden werden.

Prinzipiell ist das Verarbeitungsmodell ein beliebiger maschineller Beweiser. Welches Beweisverfahren ein Beweiser verwendet, ist für das Ergebnis irrelevant. Algorithmische Abstraktion spielt in logischen Sprachen keine Rolle, weil bereits eine maximale Abstraktion bei der Formulierung des Programms erfolgen muß. In der Praxis ist dies jedoch nur eine Idealvorstellung. Prolog als wichtigste logikorientierte Programmiersprache spezifiziert zumindest, daß Backtracking als Kontrollstruktur des Beweisverfahrens eingesetzt wird. Zusätzliche Konstrukte ermöglichen, Einfluß auf die Programmausführung zu nehmen. Beispielsweise können Zweige des Backtracking-Baums mit dem Operator `cut` abgeschnitten werden. Ihr unvorsichtiger Gebrauch kann jedoch die Vorteile deklarativer Programmierung ins Gegenteil verkehren. Deklarative Programmiersprachen spielen in der Künstlichen Intelligenz, insbesondere in der Wissensrepräsentation und im Maschinellen Lernen, eine wichtige Rolle. Neuere Konzepte aus diesem Bereich integrieren bereits im Kern ihres Verarbeitungsmodells die Paradigmen der funktionalen, logischen, nebenläufigen, objektorientierten und der Constraint-Programmierung. Hier ist die am DFKI in Saarbrücken entwickelte Sprache Oz [Smolka, 1995] zu nennen.

Die Meinungen, was denn eigentlich den *objektorientierten Programmierstil* ausmacht, gehen immer noch auseinander. Ebenso die Vorstellungen vom Verarbeitungsmodell. Objektorientierte Konzepte werden im übernächsten Abschnitt ausführlich behandelt, hier soll lediglich eine generelle Einordnung versucht werden.

Fischer und Grodzinky [1994] sehen objektorientierte Sprachen als Erweiterung bzw. Generalisierung streng (und disjunkt) typisierter Sprachen, also solcher, die die Typkonformität eines Programms zur Übersetzungszeit prüfen, so daß (die meisten) Laufzeitobjekte keine Typinformation benötigen. Die Erweiterung betrifft die Typen, die nicht mehr disjunkt sein müssen: ein Typ kann Subtyp eines anderen Typs sein. Damit verbunden ist die Vererbung von Funktionen (Simula) bzw. Methoden (Smalltalk) und der dynamische Methodendispatch zur Laufzeit in Abhängigkeit vom Argumenttyp. Alle Laufzeitobjekte benötigen somit potentiell ihre Typidentität. Diese Sicht formuliert Stroustrup in [1988] etwas abstrakter: objektorientierte Programmierung entsteht aus der Kombination abstrakter Datentypen mit der Vererbung.

Eine grundsätzlich andere Sicht lag der Entwicklung von Smalltalk zugrunde. Alan Kay beschreibt es so:

> "I spent a fair amount of time thinking about how objects could be characterized as universal computers without having to have any exception in the central metaphor." [Kay, 1996, S. 528]

Seine Bestrebung hat zum *Kommunikationsmodell* geführt. Jedes Objekt ist eigenständig, es erhält von anderen Objekten Nachrichten, führt Berechnungen durch und sendet Nachrichten an andere Objekte. Meistens wird aus dieser Vorstellung abgeleitet, daß Objekte einen veränderbaren Zustand besitzen und somit Objektorientierung im Gegensatz zu

funktionaler Programmierung stünde [Louden, 1994]. Prinzipiell kann man sich aber auch vorstellen, daß Objekte keinen Zustand besitzen bzw. konstant sind und daß Veränderungen in der Welt nur durch das Entstehen neuer Objekte und das Verschwinden alter Objekte modelliert werden, wie in strikt funktionalen Sprachen. In so einem Modell spielt die Identität von Objekten keine Rolle mehr, sondern nur noch ihre Äquivalenz.

3.2 Objektorientierung: Ein Sammelsurium von Begriffen und Konzepten

Die wesentlichen Merkmale objektorientierter Programmiersprachen wurden oben bereits angesprochen. Vereinfacht kann man zwei Fraktionen im Streit um die objektorientierte Wahrheit identifizieren. Die eine (gemäßigte) sieht objektorientierte Programmierung als Weiterentwicklung vorheriger Paradigmen, insbesondere der Idee abstrakter Datentypen ergänzt um Typsubsumption bzw. Vererbung [Stroustrup, 1988]. Die (radikale) Fraktion bricht mit der Tradition und erklärt die Welt aus einer neuen Sicht: alles ist ein aktives selbständiges Objekt, Objekte tauschen miteinander Nachrichten aus [Kay, 1996].[4]

Innerhalb des Kommunikationsmodells führt die Entscheidung, ob man zwischen Klassen und Instanzen unterscheidet oder nicht, zu zwei- oder einstufigen Objektsystemen. Letztere folgen der Metapher des *Delegierens*, d. h. wenn ein Objekt eine Nachricht nicht behandeln kann, leitet es sie an seine Delegierten weiter. Vererbung im Sinne des Delegierens wird in der Regel dynamisch zur Programmausführungszeit durchgeführt. In zweistufigen Objektsystemen geht man davon aus, daß es viele gleichartige Objekte (Instanzen) gibt, die man zu Klassen zusammenfaßt und deren Gemeinsamkeiten man in sogenannten Klassenobjekten repräsentiert. Vererbung drückt hier meistens Unter- und Oberklassenbeziehungen aus und wird in der Regel zur Übersetzungs- oder zur Ladezeit eines Programms durchgeführt. Setzt man den Klassenbegriff in Analogie zum traditionellen Typkonzept in Programmiersprachen, so werden Smalltalk, das dem Kommunikationsmodell mit Klassen/Instanzen folgt, und C++, das ohne das Kommunkationsmodell auskommt, doch sehr ähnlich.

Objektorientierung geht über Programmierung im engeren Sinn weit hinaus. Sie hat inzwischen den gesamten Softwareentwicklungsprozeß erfaßt. Es wurden spezielle objektorientierte Software-Engineering-Methoden (OOSE) entwickelt, die objektorientierte Analyse (OOA), objektorientiertes Design (OOD) und objektorientierte Programmierung (OOP) und Wartung umfassen. Besonders bekannt sind die Methoden von Booch [1994], Coad/Yourdon [1994], Jacobson [1994], Rumbaugh [1991]. Ein Vergleich dieser Methoden ist in [Fichman und Kemerer, 1992] zu finden. Ich beziehe mich in dieser Arbeit im wesentlichen auf [Booch, 1991] bzw. [Booch, 1994].

In den folgenden Abschnitten werden die verschiedenen Konzepte bzw. Sprachkonstrukte *bottom-up* erläutert. Eine deutschsprachige Begriffsklärung zur objektorientierten Pro-

[4] Allerdings werden radikale Versprechen selten eingelöst: solange Aspekte wie Parallelität und Synchronisation unberücksichtigt bleiben, hat man nicht mehr als syntaktischen Zucker geschaffen. Oder anders ausgedrückt, solange Parallelität und Synchronisation keine Rolle spielen, ist die Selbständigkeit und der Nachrichtenaustausch zum Beispiel zwischen zwei Zahlen nur eine (m. E. eher hinderliche) Metapher. Parallelität und Synchronisation wurde aber z. B. in den Arbeiten von Hewitt und Agre behandelt [Hewitt *et al.*, 1973].

grammierung ist in [Stoyan und Görz, 1983] zu finden. Nun werden zunächst die übergeordneten Motive für objektorientierte Sprachkonzepte in Erinnerung gerufen.

3.2.1 Wiederverwendbarkeit und Wartbarkeit als Motivation

Als Motivation für objektorientiertes Programmieren wird immer wieder die höhere Wiederverwendbarkeit von Software-Komponenten auf der Nutzer-Seite und die bessere Wartbarkeit auf Anbieter-Seite versprochen.

Ähnliche Fortschritte im Software-Engineering wurden schon vom Konzept abstrakter Datentypen und vom Modulkonzept erwartet. Ein Problem (disjunkter) abstrakter Datentypen war, daß die bereitgestellte Funktionalität entsprechender Module meist nicht hundertprozentig paßt, so daß Anpassungen erforderlich sind. Louden [1994] nennt fünf Wege, um Software-Komponenten wiederverwendbar zu machen: Erweiterung, Einschränkung, Redefinition, Abstraktion und Polymorphisierung.

Um die Wartbarkeit von Software-Komponenten zu erhöhen, versucht man den Zugriff auf Implementierungsdetails von außen zu verhindern. Entsprechende Konzepte zur Modularisierung, Datenkapselung und Information-Hiding gab es bereits in nicht objektorientierten Sprachen wie Modula-2 oder Ada. Diese Errungenschaften will man nun in objektorientierten Sprachen, die sich zunächst um die Anpaßbarkeit sorgten, nicht aufgeben. Es entsteht der Konflikt, nur so viele Details offenzulegen, wie für die Anpaßbarkeit nötig, und gleichzeitig möglichst viele Details zu verbergen, um Implementierungsänderungen schnittstellenneutral vornehmen zu dürfen.

Die Verquickung beider Bestrebungen in einem einzigen Sprachkonzept und einem syntaktischen Konstrukt in C++, Eiffel, Java u.a. hat m. E. nicht zu klaren Lösungen beigetragen.

3.2.2 Objekte, Klassen und Instanzen

Booch [1991, Seite 77] definiert ein Objekt folgendermaßen:

> "An object has a state, behavior, and identity; the structure and behavior of similar objects are defined in their common class; the terms instance and object are interchangeable."

Diese Vorstellung von Objekten wird als das klassenbasierte (engl. *classical*) Objektmodell betrachtet [Coplien, 1992, S. 280]. Sie liegt auch dieser Arbeit zu Grunde. Weitere ähnliche Definitionen des Objektbegriffs findet man in [Cox, 1991, S. 54], [Martin und Odell, 1992, S. 241], [Rumbaugh, 1991], [Shl, , S.14]. Die Struktur von Objekten wird über seine Instanzvariablen (bzw. Attribute, Slots, Felder oder Members) definiert, das Verhalten über Methoden.

Objektsysteme, die zwischen Klassen und Instanzen unterscheiden werden auch als zweistufig bezeichnet, im Gegensatz zu sogenannten einstufigen Systemen, die diese Unterscheidung nicht vornehmen. Letztere folgen dem Modell des Delegierens, dessen bekannteste Vertreterin die Sprache Self ist [Chambers *et al.*, 1989](s. 3.2.5). Werden Klassen als Objekte erster Ordnung geführt, so sind sie selbst Instanzen einer Klasse, die dann als Metaklasse bezeichnet wird. Man kann sich nun vorstellen, daß die Instanziierungsrelation

beliebig tief ist und man somit ein n-stufiges Modell bzw. eins mit unendlich vielen Stufen erhält. In Smalltalk werden drei Stufen unterschieden: (terminale) Instanzen, Klassen und Metaklassen, wobei jeder Klasse eine eigene Metaklasse zugeordnet wird. In CLOS sind Klassen zwar auch Objekte erster Ordnung, man kommt aber mit zwei Stufen aus, weil Klassen und Metaklassen uniform behandelt werden und man den Trick einer selbstinstantiierten Klasse anwendet, d. h. es gibt eine ausgezeichnete Klasse, die Instanz von sich selbst ist und somit sowohl als einfache Klasse als auch als Metaklasse gesehen werden kann. Erweiterbare objektorientierte Programmiersprachen wie CLOS MOP und ΤΕΛΟΣ erlauben auch benutzerdefinierte Metaklassen. Dies ist aber bisher eher die Ausnahme.

3.2.3 Kapselung, Information-Hiding und Modularisierung

Der Wunsch nach Kapselung von Objekten leitet sich unmittelbar aus den Vorstellungen des Kommunikationsmodells ab. Objekte als selbständig agierende Einheiten brauchen eine schützende, nach außen abgrenzende Hülle. Nach [Booch, 1991, Seite 77] bedeutet Kapselung folgendes:

> "Encapsulation is the process of hiding all of the details of an object that do not contribute to its essential characteristics."

Mit dieser Definition wird gleich deutlich, daß Kapselung hier auf Datenstrukturen bzw. Objekte bezogen wird. Streng genommen müßte man also von Datenkapselung sprechen. Diese ist abzugrenzen von wohl bekannten Mitteln der Programmkapselung oder auch Blockstrukturierung mit entsprechenden Sichtbarkeitsregeln, Gültigkeitsbereichen und der Lebensdauer von Variablen- und Konstantenbindungen. Letztere fanden eine konsequente Realisierung in der Sprache Modula-2 [Wirth, 1985a]. Dort wurde auf den Punkt gebracht, worum es primär geht, nämlich um die funktionale Modularisierung von Programmen in mittelgroße Einheiten, die über wohldefinierte Schnittstellen miteinander interagieren. Die Schnittstellen enthalten genaue Import- und Exportspezifikationen. Will man diese Konzepte und Lösungen in einer Programmiersprache beibehalten, so ist es naheliegend den objektorientierten Teil dazu orthogonal zu entwerfen und ihn nicht zusätzlich mit Kapselungsaspekten zu überfrachten. Diese Position wurde konsequent in EuLisp vertreten.

Wichtiger Streitpunkt der Sprachdesigner ist die Frage, ob Sichtbarkeitsbeschränkungen von der Programmiersprache erzwungen werden sollen oder ob nur unbeabsichtigte Zugriffe durch zufällig gleiche Namen vermieden werden sollen. Noch allgemeiner manifestiert sich hier der Gegensatz, ob eine Sprache primär etwas Negatives verbieten soll (Mißbrauch, Fehler etc.) oder etwas Vernünftiges ermöglichen soll. Lisp folgt traditionell dem letzteren Prinzip, die meisten statischen Programmiersprachen dem ersten.

Da der Durchbruch objektorientierter Ideen im wesentlichen mit Smalltalk kam und man dort in einem gewissen Übergeneralisierungseifer alles zum Objekt und vom Objekt her erklären wollte, mußten Aspekte der Modularisierung sowie Sichtbarkeitsregeln und Sprachkonstrukten zur Klassifizierung von Objekten verschmolzen werden. So sind denn auch die Kapselungsmechanismen eher einfach ausgeprägt: Instanzvariablen sind nur in Methoden der definierenden Klasse und in Methoden der Subklassen sichtbar, nicht aber in Methoden anderer Klassen. Alle Methoden einer Klasse sind nach außen sichtbar, jedoch sollen solche, die als privat deklariert sind, nicht von außerhalb der Klasse aufgerufen werden. Daß

Smalltalk hier nur einfache Mittel bereitstellt hängt aber auch damit zusammen, daß es als dynamische Sprache im Unterschied zu C++ keine Typprüfungen von Operationsaufrufen (Sendeereignissen) zur Übersetzungszeit vornimmt.

C++, Eiffel [Meyer, 1992] und auch Java [Gosling *et al.*, 1996] folgen Smalltalk in der Entscheidung, daß Klassen gleichzeitig auch die Rolle von Modulen spielen sollen. C++ stellt einen allgemeinen Mechanismus zur Datenkapselung bereit, der es erlaubt zwischen den Kategorien *public*, *protected* und *private* zu unterscheiden. Dabei bedeutet *public*, daß solche Klassenelemente (engl. *members*) von überall sichtbar sind, *protected* bedeutet, daß solche Elemente in der Klasse und in Subklassen sichtbar sind, und die als *private* gekennzeichneten Elemente sind schließlich nur in der Klassendefinition selbst sichtbar.

Man muß aber erwähnen, daß Stroustrup selbst deutlich zwischen Klassen und Modulen differenziert:

> ... Data hiding and a well-defined interface can also be obtained through a module concept. However, a class is a type; to use it, one must create objects of that class, and one can create as many such objects as are needed. ... [Stroustrup, 1987]

Und auch Wegner [1987] sieht das Innovative der Objektorientierung eher in ihrer Eigenschaft, das Klassifizieren zu unterstützen, als in der Erweiterung des Modulkonzepts. Ebenso plädiert Szyperski [1992] für die Differenzierung und Beibehaltung beider Konzepte: der Module und der Klassen.

3.2.4 Vererbungskonzepte

Die sogenannte Vererbung ist ein Schlüsselkonzept objektorientierter Programmierung. So werden in [Wegner, 1987] Programmiersprachen, die zwar Klassen und Methoden zur Verfügung stellen, aber keine Vererbungsmechanismen, als objektbasiert und nicht als objektorientiert bezeichnet. Ähnlich sieht es [Stroustrup, 1988, S. 13,18] und auch Booch [1991, Seite 96]:

> "Inheritance is perhaps the most powerful of these relationships and may be used to express both generalization and association."

So einig man sich in der Einschätzung der Bedeutung von Vererbungskonzepten ist, so unterschiedlich sind die konkreten Mechanismen und ihr intendierter Verwendungszweck in verschiedenen Programmiersprachen. Begrifflich gehen Vererbungskonzepte auf Vorstellungen aus der Genetik und des Zivilrechts zurück, also auf das Erben von Genen, Konventionen, Eigentum, Verpflichtungen, die von den Eltern auf die Kinder übergehen bzw. übertragen werden. So fallen im Zusammenhang mit Vererbungskonzepten in Programmiersprachen auch immer wieder Begriffe wie "Vaterklasse", "Sohnklasse" oder, weniger patriarchalisch, *parent* und *child*; und das an Stellen, wo man die Oberklasse bzw. Unterklasse meint. In meinen Augen zeigt die "Vererbung" besonders deutlich, daß die unreflektierte Verwendung von Metaphern auch verwirrend sein kann.

Neben Simula [Dahl und Nygaard, 1966] hat die objektorientierte Programmierung ihre Wurzeln in der objektorientierten Wissensrepräsentation, insbesondere den semantischen

Netzen [Quillian, 1967], [Carbonell, 1970], [Winston, 1975], den Frame-Sprachen [Minsky, 1975], [Bobrow und Winograd, 1977], [Roberts und Goldstein, 1977] und den Aktor-Sprachen ACTOR [Hewitt *et al.*, 1973], Act1 [Lieberman, 1981]. Bei [Quillian, 1967] ging es um die Frage der Modellierung des Wissens im Gedächtnis eines Menschen. Daran orientierte sich auch der Vererbungsmechanismus von Begriffseigenschaften, der eher das Assoziieren von Wissenselementen als die Generalisierung und Spezialisierung von Begriffen unterstützte. So gehen einstufige Objektsysteme (ohne die Klasse/Instanz-Unterscheidung) sowie das Konzept der *shared slots* in klassenbasierten Programmiersprachen letztlich auf die Ideen der Gedächtnismodellierung zurück, ohne daß man sich heute dessen im allgemeinen bewußt ist.

Der Hauptstrang der verschiedenen Vererbungskonzepte rankt sich jedoch um die Ähnlichkeitsbeziehung zwischen Klassen. Dies führt auf der Modellierungsseite zu einem besseren Verständnis von Anwendungsdomänen und ihrer Komplexitätsreduktion sowie auf der Implementierungsseite zur Wiederverwendung, Erweiterung und einfacheren Anpassung von Softwarebausteinen. Darauf ausgerichtete Vererbungskonzepte zielen also direkt auf eine signifikante Verbesserung und Beschleunigung der Software-Entwicklungsprozesse.

Beim Modellieren geht man üblicherweise *bottom-up* vor, d. h. man geht von konkreten Objekten mit ihren Eigenschaften aus, bildet Klassen, analysiert die Klassen auf Ähnlichkeiten, generalisiert gemeinsames in Oberklassen usw. Beim Implementieren ist es, zumindest bei der Ergebnisbetrachtung eines Programms als sequentieller Text, eher umgekehrt. Zunächst werden einfache und generelle Objekteigenschaften in Klassendefinitionen spezifiziert, dann werden speziellere Klassen mit mehr und speziellerer Funktionalität als Subklassen der Ausgangsklassen definiert. Schließlich werden, angestoßen durch das Hauptprogramm, konkrete Instanzen einiger Klassen erzeugt und manipuliert.

Je nach dem, welchen Akzent Sprachdesigner setzen möchten, wählen sie eine entsprechende Terminologie im Zusammenhang mit der Vererbung:

- *superclass* vs. *subclass* in Smalltalk, CLOS, Java etc.;
- *base class* vs. *derived class* in C++;
- *ancestor* vs. *descendant* in Eiffel;
- *parent* vs. *child* in einstufigen Objektsystemen;

Das Definieren einer Subklasse wird oft als *Spezialisieren*, *Erweitern* oder auch neutral als *Subclassing* bezeichnet, während der umgekehrte Vorgang des (Heraus-) Faktorisierens von Gemeinsamkeiten in eine Superklasse als *Generalisieren* bezeichnet wird. Das *Overriding* wird im Deutschen oft nicht ganz zutreffend mit Überschreibung übersetzt und bedeutet, daß eine in der Superklasse bereits definierte Methode in der Subklasse nochmal definiert wird, was dazu führt, daß beim Anwenden dieser Operation auf eine Instanz der Subklasse auch die Methode der Subklasse zum tragen kommt und nicht die der Superklasse. Dies auch dann, wenn der speziellste Typ (oder Klasse) eines Objekts (einer Instanz) zur Übersetzungszeit nicht ableitbar ist. Hier spricht man auch vom Laufzeitdispatch.

Bezüglich der Instanzvariablen ist in vielen Sprachen wie Smalltalk, C++, Java ein *Overriding* nicht möglich. In Smalltalk ist es ein Fehler, eine Instanzvariable mit dem gleichen

Namen wie in einer Superklasse zu spezifizieren. In C++ und Java führt dies zu einer zweiten Instanzvariablen mit dem gleichen Namen. Für speziellere Methoden ist nur die zweite sichtbar, während für die Methoden der Superklasse nur die erste Instanzvariable sichtbar ist. Diese Strategie wird in Java *Shadowing* genannt. Das *Overriding*, als ein Mechanismus, die speziellsten Eigenschaften eines Objekts zur Laufzeit wirken zu lassen, ist aber eine notwendige Voraussetzung, um Struktur und Verhalten von Objekten adäquat und möglichst deklarativ zu spezialisieren.

Je nachdem ob eine Klasse nur eine oder mehrere direkte Superklassen haben darf, spricht man von *einfacher Vererbung* oder von *multipler Vererbung*. Eine spezielle Form der multiplen Vererbung ist die *Mixin-Vererbung* [Bretthauer *et al.*, 1989a]. Dabei werden Eigenschaften substantivischen Charakters nach einfacher Vererbung klassifiziert. Zusätzliche adjektivische Eigenschaften können nach multipler Vererbung in sogenannten *mixins* oder Mixin-Klassen klassifiziert werden. Eine substantivische Klasse kann also mehrere Mixin-Klassen und genau eine substantivische Klasse spezialisieren.

Eine Formalisierung der Vererbungskonzepte findet man in [Touretzky, 1986]. Praktische Probleme multipler Vererbung werden in [Ducournau und Habib, 1991], [Ducournau *et al.*, 1992] und [Bretthauer *et al.*, 1989b] diskutiert. Vererbungskonzepte werden im nächsten Kapitel noch ausführlicher behandelt.

3.2.5 Delegieren

Vereinfacht kann man sagen, daß Delegieren in einstufigen Objektsystemen, wo es keine Unterscheidung zwischen Klassen und Instanzen gibt, dem entspricht was Vererbung in zweistufigen Objektsystemen ausmacht. In letzteren organisiert man ein Programm in einer Generalisierungs- bzw. Spezialisierungshierarchie, während man in Self [Smith und Ungar, 1995] zunächst allgemein eine Arbeitsteilung zwischen verschiedenen Objekten mit jeweiligen Verantwortlichkeiten vornimmt. Bestimmte Aufgaben werden an bestimmte Objekte *delegiert*. Implizit kann man in Self natürlich auch Spezialisierungshierarchien ausdrücken, diese werden aber nicht syntaktisch und semantisch unterstützt. Da Delegierungssprachen nicht zur Klassen-/Instanz-Trennung zwingen, werden sie häufig als mächtiger und ausdrucksstärker im Vergleich mit zweistufigen Objektsystemen bezeichnet. Dem ist aber entgegenzusetzen, daß ein flexibleres bzw. universelleres Konzept nicht notwendigerweise einen bestimmten Programmierstil unterstützt. In der Regel läßt es mehr zu, unterstützt aber weniger. Insofern sehe ich im Konzept des Delegierens keine Alternative zu zweistufigen Objektsystemen, die das Klassifizieren, Spezialisieren und Generalisieren von Objekten unterstützen. Delegieren unterstützt vielmehr eine spezielle Ausprägung des Kommunikationsmodells.

3.2.6 Senden von Nachrichten

Wie schon oben erwähnt liegt dem objektorientierten Programmierstil nach Smalltalk das *Kommunikationsmodell* zugrunde. In diesem Modell tauschen die Kommunikationspartner, genannt *Objekte*, Informationen durch *Versenden von Nachrichten* aus. Berechnungen erfolgen dadurch, daß Objekte Nachrichten, die sie erhalten, verarbeiten, wobei sie weitere Nachrichten an andere Objekte oder an sich selbst versenden. Jedes Objekt hat

in der Regel einen inneren Zustand, der von außen nicht direkt einsehbar ist, außer über Nachrichten. Es gibt also ein Sprachkonstrukt der Art:

receiver method-name $argument_1$ ...

mit der Semantik, daß zuerst die Klasse (Typ) von *receiver* bestimmt wird[5] und dann die dieser Klasse bezüglich *method-name* zugeordnete Methode auf die gegebenen Argumente angewendet wird.

Solange Parallelität keine Rolle spielt und das Nachrichtenversenden sequentiell erfolgt entspricht ein Sendeereignis einem Prozeduraufruf in imperativen Programmiersprachen bzw. einer Funktionsapplikation in funktionalen Sprachen. So wird der obige Ausdruck in Java folgendermaßen notiert:

receiver.method-name($argument_1$`, ...);`

In CLOS, das auf dem primär funktionalen CommonLisp gründet, hat man daher das Nachrichtenversenden unter der Funktionsapplikation subsummiert, mit folgender Syntax:

`(`*method-name receiver* $argument_1$ `...)`

Die mit *method-name* assoziierte Operation bekommt einen eigenständigen Charakter und wird als *generische Funktion* bezeichnet, der Empfänger wird zum ersten Argument, das ursprünglich erste Argument wird zum zweiten Argument usw. CLOS geht noch einen Schritt weiter und erlaubt, die Methodenauswahl von allen Argumente bzw. deren Klasse abhängig zu machen. Man spricht daher auch von *Multimethoden*. Generische Operationen und Methodenauswahl werden im nächsten Kapitel noch ausführlicher behandelt.

3.2.7 Überladen von Operatoren und Polymorphie

Schon die ersten höheren Programmiersprachen erlaubten, einige Operatoren auf Argumente mehrerer Typen anzuwenden. Typischerweise können die vordefinierten arithmetischen Opertoren `+`, `-` etc. sowohl auf ganze als auch auf reelle Zahlen angewandt werden[6]. Dabei muß der Übersetzer schon syntaktisch erkennen, welche Operation gemeint ist, und diese einsetzen. In Ada [Nagl, 1982] wurde die Möglichkeit, *Operatoren zu überladen* (engl. *operator overloading*), auch dem Benutzer eingeräumt. Für benutzerdefinierte Typen A und B kann er folgende Funktionen definieren:

```
function + (x, y in A) return A;
function + (x, y in B) return B;
```

Es wird also ein Name (hier `+`) mit mehreren Operationen assoziiert. Aus dem Verwendungskontext muß aber schon zur Übersetzungszeit eindeutig hervorgehen, welche Operation gemeint ist. *Overloading* ist ein rein syntaktisches Konzept und wird auch als *ad-hoc Polymorphie* bezeichnet.

Ein eher semantisches Konzept ist das der *polymorphen Operationen*. Dabei kann eine Operation auf Argumente mehrerer Typen angewandt werden. Die genauen Argumenttypen müssen zur Übersetzungszeit nicht notwendigerweise bekannt sein. Solche Operationen

[5] Dies entfällt in einstufigen Objektsystemen.

[6] Natürlich sind hier streng genommen nicht die mathematischen Begriffe gemeint, sondern ihre Entsprechungen in einer Programmiersprache.

werden auch *generisch* genannt. Auch Datentypen bzw. Klassen können *generisch* bzw. *parametrisiert* sein, z. B. *template classes* in C++ und Eiffel oder Typen in Ada:

```
In C++: template<class T> class Queue ... T item; ...
In Ada: type Stack(Size: POSITIVE) is record ...
```

Für generische Operationen und Datentypen wird auch der Begriff der *parametrisierten Polymorphie* verwendet, wenn explizit oder implizit ein Typparameter die konkrete Operationsanwendung bzw. den konkreten Datentyp bestimmt. Generell geht der in der Informatik verwendete Begriff der *Polymorphie* auf Strachey [1967, S.xx] zurück:

> "Parametric polymorphism is obtained when a function works uniformly on a range of types; these normally exhibit some common structure. Ad-hoc polymorphism is obtained when a function works, or appears to work on several different types (which may not exhibit a common structure) and may behave in unrelated ways for each type."

Bezogen auf objektorientierte Sprachen verdeutlicht Booch mit folgender Definition, daß dieses Konzept eng mit der obligatorischen Typisierung von Variablen in traditionellen höheren Programmiersprachen zusammenhängt [1991, S.517]:

> "... A concept in type theory, according to which a name (such as a variable declaration) may denote objects of many different classes that are related by some common superclass; thus, any object denoted by this name is able to respond to some common set of operations in different ways ...,"

Meyer [1988] bringt es mit seiner Bemerkung noch spitzer auf den Punkt:

> "... there is no way the type of an object can ever change. Only a reference can be polymorphic ..."

Meyer läßt aber außer Acht, daß in dynamischen objektorientierten Sprachen wie CLOS Objekte auch ihre Typ- bzw. Klassenzugehörigkeit ändern können (mit `change-class`). Schon in CLOS erfordert diese Möglichkeit spezielle Vorkehrungen in der Sprachspezifikation und in ihrer Implementierung. Mit *Compile-Time-Type-Checking-Sprachen* dürfte `change-class` im allgemeinen kaum vereinbar sein. Klassenwechsel müßten auf solche vom Allgemeinen zum Speziellen entlang der Spezialisierungsbeziehungen beschränkt bleiben. Eine ausführliche Diskussion von Aspekten der Polymorphie ist in [Cardelli und Wegner, 1985] zu finden.

3.2.8 Methodenkombination

Alle objektorientierten Programmiersprachen bieten die Möglichkeit Methoden der Superklassen zu erben, die meisten erlauben auch, speziellere Methoden in Subklassen zu definieren, was dann als *Redefinition* bezeichnet wird. Soweit können auch Module aus Modula-2 mithalten. Prinzipiell neu ist bei objektorientierten Sprachen die dynamische

(zur Laufzeit) Auswahl der speziellsten Methode gemäß der speziellsten Klasse des aktuellen Empfängers (oder Parameters in CLOS), was auch *dynamic binding* genannt wird. Dies kann mit Modulen aus Modula-2 oder Packages aus Ada nicht erreicht werden.

Oft möchte man nicht alles reimplementieren, was in der Supermethode schon implementiert ist, sondern nur einige Ergänzungen bzw. Modifikationen vornehmen. Aus einer übergeordneten Sicht möchte man das Gesamtverhalten (bzgl. einer Operation) so entlang der Superklassenhierarchie konstruieren, daß nichts doppelt oder mehrfach implementiert und gewartet werden muß, sondern an genau einer Stelle. Dafür reicht das herkömmliche Mittel der prozeduralen Dekomposition, auf die sich C++ an dieser Stelle zurückzieht[7], nicht aus. Daher bieten z. B. Smalltalk, Java und CLOS einen abstrakteren Mechanismus, in einer spezielleren Methode die nächst allgemeinere Methode auszuführen. Beispielkonstrukte sind `call-next-method` in CommonLisp, `run-super` in Smalltalk und `super()` in Java. Die Sprache BETA [Madsen *et al.*, 1993, S. 94] bietet einen umgekehrten Mechanismus an: das Konstrukt `inner` erlaubt, in einer Methoden die nächst speziellere Methode aufzurufen.

In CLOS wurde das ausdrucksstärkste Konzept der *Methodenkombination* entwickelt, das zusätzlich zu *Primärmethoden* sogenannte *Before-*, *After-* und *Around-Methoden* anbietet. Dabei werden, falls vorhanden, zuerst alle Before-Methoden (die spezielleren zuerst), dann die speziellste Primärmethode (über `call-next-method` evtl. auch weitere) und schließlich alle After-Methoden (die speziellste zuletzt) ausgeführt. Around-Methoden sind eine Art Schlinge, die sich um die gewachsene Zwiebel aus Before-, Primär-, und After-Methoden legen können, um die Kontrolle über die Zwiebel zu erhalten und sie ggf. unwirksam werden zu lassen. Siehe hierzu auch Abschnitt 3.4.3 in diesem Kapitel.

3.2.9 Reflektion und Metaobjektprotokolle

Grundlegende Arbeiten zur *Reflektion in Programmiersprachen* begannen mit [Smith, 1982], [Smith, 1984]. Im Prinzip kann man auch Rekursion als eine Form des Selbstbezugs und somit der Reflektion ansehen. Über Programmiersprachen hinaus wurden *reflektive Systeme* insbesondere in der Künstlichen Intelligenz untersucht [Maes, 1987] [Maes und Nardi, 1988], [Karbach, 1994]. Aktuelle Arbeiten aus beiden Strängen wurden in [Yonezawa und Smith, 1992] und [Kiczales, 1996] vorgestellt.

Metaobjektprotokolle (MOPs) verfolgen ein spezielles Ziel im Rahmen der Reflektion in Programmiersprachen und Systemen. Während Vererbungskonzepte objektorientierter Programmiersprachen die Wiederverwendung (durch Spezialisierung) von erstellten Anwendungen unterstützen, zielen Metaobjektprotokolle auf die Wiederverwendung der Programmiersprache selbst [Kiczales *et al.*, 1991], [Bretthauer *et al.*, 1993], indem die Sprache spezialisiert wird. Diesen qualitativen Sprung kann man mit dem Schritt von fest programmierten Automaten zu programmierbaren universellen Rechnern vergleichen. Allerdings ist damit zu rechnen, daß diese zugegeben provokative These erst in einigen Jahren, wenn nicht Jahrzehnten, erhärtet werden kann. Immerhin sprießen inzwischen diverse Ableger von CLOS MOP und Τελος im Umfeld von Oberon [Templ, 1994], BETA [Brandt, 1995], ja sogar von C++ [Buschmann *et al.*, 1992].

[7] Nach dem Motto, was kümmert mich objektorientierte Programmierung? Wenn man schon zu Fuß den dynamischen Speicherbedarf verwaltet, dann ist man dankbar, auch weiterhin strukturiert programmieren zu müssen. Dann fällt es auch leichter, sich an das neue Paradigma zu gewöhnen!

JAVA hat mit JDK1.1 den ersten Schritt in Richtung MOP getan und stellt ein umfassendes *Introspektionsprotokoll* im *Reflection package* zur Verfügung. Introspektionsprotokolle erlauben, ein beliebiges Objekt, seine Klasse und somit alle wesentlichen Elemente, die die Struktur und das Verhalten des Objekts bestimmen, zu inspizieren. Klassen, Methoden und Felder (Klassen- und Instanzvariablen in Java) sind somit auch Objekte, sie werden als *Metaobjekte* bezeichnet. Generische Werkzeuge wie Klassenbrowser, Inspektoren, Debugger etc. können somit portabel implementiert werden.

Metaobjektklassen sind in Java *final* und können nicht spezialisiert werde, wie in CLOS MOP und ΤΕΛΟΣ, wo neuartige Klassen, z. B. mit multipler Vererbung, oder multiple Methoden, deren Argumente den Methodendispatch mitsteuern, oder spezielle Annotationen von Instanzvariablen als Spracherweiterung realisiert werden können. Entsprechende Teilprotokolle legen offen,

- wie die Erzeugung und Initialisierung systemdefinierter Metaobjekte zu geschehen hat und wie diese gezielt spezialisiert werden können;
- wie die Linearisierung der Vererbungshierarchie durchgeführt wird;
- wie Instanzvariablen mit ihren Annotationen vererbt werden;
- wie Instanzvariablenzugriffe realisiert werden, bzw. wie entsprechende Lese- und Schreiboperationen generiert werden;
- wie für eine gegebene Signatur die Auswahl der anwendbaren Methoden geschieht, wie sie sortiert werden, und welche schließlich in welcher Reihenfolge auszuführen sind.

Diese Aspekte werden am Beispiel CLOS MOP und in späteren Kapiteln ausführlich behandelt und bilden einen Schwerpunkt dieser Arbeit.

3.3 Ausgewählte verwendete Objektsysteme

In diesem Abschnitt werden exemplarisch einige der wichtigsten und für diese Arbeit relevanten objektorientierten Programmiersprachen vorgestellt:

- Smalltalk als das Original, das die Entwicklung maßgeblich beeinflußt hat,
- C++ als die heute am meisten verwendete objektorientierte Sprache,
- Java, als die modernere, internet-taugliche Variante von C++ und
- CLOS, MFS und ObjVLisp als die konkrete Ausgangsbasis für diese Arbeit.

Man könnte sagen, daß sich JAVA zu C++ so verhält, wie ΤΕΛΟΣ zu CLOS, jedenfalls was den objektorientierten Teil der Sprachen angeht.

Exemplarisch gehe ich auch auf die jeweilige historische Entwicklung und die getroffenen Designentscheidungen ein. Punktuell werden natürlich auch weitere Programmiersprachen angesprochen wie DYLAN [Shalit, 1996], BETA [Madsen *et al.*, 1993], Oberon [Wirth,

1985b], Eiffel [Meyer, 1992] etc. Ihre ausführliche Darstellung würde aber den Rahmen dieser Arbeit sprengen.

Ein konzeptueller Vergleich der wichtigsten objektorientierten Programmiersprachen ist in [Kristensen, 1996] zu finden. Diskussionsbeiträge aus der Sicht der Entwickler von Anwendungen der Künstlichen Intelligenz wurden in [Voß, 1995] zusammengestellt.

3.3.1 Smalltalk

Smalltalk [Goldberg und Robson, 1983], [Goldberg und Robson, 1989] entstand im Rahmen des Forschungsprojekts *Dynabook*, das unter der Leitung von Allen Kay Anfang der 70er Jahre bei Xerox PARC durchgeführt wurde. Maßgeblichen Einfluß auf das Sprachdesign hatten neben anderen Adele Goldberg und Daniel Ingalls. Das visionäre Ziel des Gesamtprojekts war das *personal computing*, in Abkehr vom damals vorherrschenden *mainframe computing*. Dabei wurden grundlegend neue Konzepte wie Fensteroberflächen, graphische Displays und Benutzerschnittstellen, Zeigeinstrumente und objektorientiertes Programmieren entwickelt. So kann auch Smalltalk im Ergebnis weniger als eine Programmiersprache im bis dahin üblichen Sinn verstanden werden, sondern vielmehr als ein Gesamtsystem, das Adele Goldberg und David Robson [1983, S. vii] kurz so charakterisieren:

- Smalltalk is a vision,
- Smalltalk is based on a small number of concepts, but defined by unusual terminology,
- Smalltalk is a graphical, interactive programming environment,
- Smalltalk is a big system.

Auch das folgende Zitat von Allen Kay [1996, S. 596] betont den Systemcharakter von Smalltalk und nennt retrospektiv sein auszeichnendes Merkmal:

> "I think the thing ... that was the most important reason to use a language like Smalltalk or Lisp, is the meta-definition facility. You should not take the language any more as a collection of features done by some designer, but you should take it as a way of enabling the language that you actually need, to write a system."

Dieser Gesichtspunkt ist aber in Lisp, insbesondere in CLOS MOP und ΤΕΛΟΣ noch wesentlich bewußter und gezielter berücksichtigt worden als in Smalltalk, das weder eine syntaktische Erweiterung zuläßt, noch beispielsweise die Spezialisierung der Vererbung oder des Methodendispatches.

Zum Gesamtsystem Smalltalk gehören der Compiler, der Interpretierer, das Laufzeitsystem und die Programmentwicklungsumgebung mit einem Browser, Inspektor, Editor und GUI-Werkzeug. Wie weit Smalltalk damit seiner Zeit voraus war zeigt folgende Episode [Kay, 1996, S.560]:

> "... it was already 1979, and we found ourselves doing one of our many demos, but this time for a very interested audience: Steve Jobs, ... from Apple. They had started a project called *Lisa* ... Thus, more than eight years after overlapping windows had been invented and more than six years after the ALTO[8] started running, the people who could really do something about the ideas finally got to see them ... Steve Jobs said he did not like the blt-style[9] scrolling we were using and asked if we could do it in a smooth continuous style. In less than a minute Dan[10] found the method involved, made the (relatively major) changes, and scrolling was now continuous! This shocked the visitors, especially the programmers among them, as they had never seen a really powerful incremental system before."

Neben der Inkrementalität des Entwicklungsprozesses hat die Portabilität von Smalltalk-Anwendungen einschließlich ihrer Benutzerschnittstellen maßgeblich zur hohen Popularität des Systems beigetragen. In beiden Aspekten hat Java, im Kontext von Internet-Anwendungen, sich Smalltalk zum Vorbild genommen.

Zu den grundlegenden Sprachdesignentscheidungen gehören folgende:

- Als Verarbeitungsparadigma dient das Kommunikationsmodell mit nachrichtenaustauschenden Objekten.
- Es wird zwischen Klassen und Instanzen unterschieden.
- Es gib nur einfache Vererbung zwischen Klassen; alle Instanz- und Klassenvariablen sowie alle Methoden einer Klasse werden an die Subklassen vererbt.
- Klassen ersetzen vollständig das Typkonzept, es gibt keine primitiven Typen außerhalb der Klassenhierarchie.
- Keine statische Typisierung von temporären sowie von Instanz- und Klassenvariablen, außer der speziellen Variable `self`, die in Methoden immer an das Empfängerobjekt gebunden ist. Daraus ergibt sich entsprechend die dynamische Typprüfung und der Methodendispatch zur Laufzeit.
- Einfache Methodenkombination über die spezielle Variable `super`.
- Metaklassen und Reflektion, auch Quellcode-Interpretation, inkrementelle Kompilation, *lazy evaluation* von Blöcken.
- Kein Metaobjektprotokoll, keine syntaktische Erweiterbarkeit (keine Makros).

Zusammenfassend kann man festhalten, daß Smalltalk eine herausragende Rolle bei der Etablierung objektorientierter Programmierkonzepte gespielt hat. Durch die ganzheitliche Sicht hat Smalltalk nicht nur im Sprachdesign, sondern auch im gesamten Software-Entwicklungsprozeß neue Maßstäbe gesetzt. Auch sein *Model-View-Controller-Konzept*[11]

[8] Der bei Xerox PARC entwickelte PC.

[9] Die Bedeutung ist hier irrelevant.

[10] Dan Ingalls, Hautimplementierer von Smalltalk bei Xerox PARC.

[11] Danach werden Softwaresysteme mit Benutzerinteraktionskomponenten grundsätzlich nach drei Aspekten strukturiert und entworfen: dem funktionalen Modell, seiner Visualisierung und der Steuerung von Benutzerinteraktionen. Nicht alle Smalltalk-Systeme unterstützen aber das Model-View-Controller-Konzept.

ist beispielgebend für den modernen Entwurf von Softwaresystemen. Die künftige Entwicklung und Verwendung von Smalltalk für komplexe Anwendungen wird nach meiner Einschätzung am stärksten davon beeinflußt, wie erfolgreich Java letztlich sein wird. Im Vergleich mit dynamischen objektorientierten Sprachen wie CLOS, DYLAN und ΤΕΛΟΣ fehlen Smalltalk die syntaktische Erweiterbarkeit, die Unterstützung anderer Programmierstile und das erweiterbare Metaobjektprotokoll. Im Vergleich mit C++ wird Smalltalk in performanzkritischen Bereichen schwächer bleiben. Java bietet gegenüber Smalltalk die Vorteile, daß es *Compile-Time-Type-Checking* und Internet-Anwendungen unterstützt.

3.3.2 C++

C++ wurde von Bjarne Stroustrup seit 1979 bei AT&T Bell Labs, wo auch C entstand, entwickelt. Sein anfängliches Ziel war es, die objektorientierten Sprachmittel aus Simula mit der Effizienz und Flexibilität von C für den Bereich der Systemprogrammierung zu verbinden. Er beabsichtigte, das Ergebnis innerhalb eines halben Jahres für konkrete Projekte nutzen zu können. Stroustrup beschreibt es retrospektiv in [1994, S.1]:

> "At the time, mid-1979, neither the modesty nor the preposterousness of that goal was realized. The goal was modest in that it did not involve innovation, and preposterous in both the time scale and its Draconian demands on efficiency and flexibility. While a modest amount of innovation did emerge over the years, efficiency and flexibility have been maintained without compromise. ... C++ as used today directly reflects its original aims."

Der Erfolg gibt ihm recht. Heute ist C++ die meistverwendete (objektorientierte) Programmiersprache überhaupt. Die wichtigsten Meilensteine der Entwicklung waren:

- 1979: *C with Classes*;
- 1983: erste verwendete C++ Implementierung;
- 1985: erste externe C++ Version, *The C++ Programming Language* [Stroustrup, 1987], CFront 1.0 als kommerzielle Version;
- 1986: *What is Object-Oriented Programming* [Stroustrup, 1988],
- 1989: CFront 2.0, Beginn der ANSI Standardisierung;
- 1990: Borland C++ release, *The Annotated C++ reference Manual* [Ellis und Stroustrup, 1990]
- 1991: CFront 3.0 (mit templates), *The C++ Programming Language* [Stroustrup, 1991];
- 1994: Draft ANSI/ISO Standard;

Aus der Diskussion seines Beitrags zur HOPL-2 [Strousrup, 1996] konstatiert Stroustrup, daß es keine allgemein akzeptierte Vorstellung über Wesen und Hauptzweck von Programmiersprachen gibt. Aus seiner Sicht muß eine universelle Programmiersprache viele Aspekte berücksichtigen, um die verschiedenen Benutzergruppen zu unterstützen. Eine Sprache ist:

- ein Tool, um Maschinen zu steuern;
- ein Kommunikationsmittel für Programmierer;
- ein Ausdrucksmittel für high-level Design;
- ein Notationsformalismus für Algorithmen;
- drückt Beziehungen zwischen Konzepten aus;
- ein Experimentierwerkzeug.

Nach Stroustrups Auffassung [1994, S.7] muß die Informatik allgemein und die Designer von Programmiersprachen insbesondere sich verschiedener Mittel aus der Mathematik, den Ingenieurwissenschaften, der Architektur, der Kunst, der Biologie, der Soziologie und der Philosophie bedienen. Eine Programmiersprache wird entworfen, damit bestimmte Problemklassen in einem gegebenen zeitlichen Kontext von bestimmten Menschengruppen gelöst werden können. Ausgehend vom ersten Entwurf wächst sie mit dem neu entstehenden Bedarf, der sich aus neuen Problemen und Lösungsmethoden ergibt.

Diese Sicht teile ich voll und ganz. Wundert man sich nun über die Divergenz der Designergebnisse in TELOS und C++, so liegt dies zum großen Teil am unterschiedlichen Kontext (KI-Programmierung vs. Betriebssystemprogrammierung) und Traditionen der jeweils beteiligten Gruppen. Um so wichtiger erscheint ein Blick über den Tellerrand eines Teilgebiets. Die Integration verschiedener Traditionen kann aber noch länger dauern als ihre Etablierung.

Stroustrup differenziert zwischen Designregeln und Prinzipien. Regeln lassen im Unterschied zu Prinzipien Ausnahmen zu. Er nennt folgende vier Gruppen von Designregeln, die für C++ angewandt wurden [Stroustrup, 1994, S.109ff]:

- allgemeine Regeln:
  - "C++'s evolution must be driven by real problems.
  - Don't get involved in a sterile quest for perfection.
  - C++ must be useful *now*.
  - Every feature must have a reasonably obvious implementation.
  - Always provide a transition path.
  - C++ is a language, not a complete system.
  - Provide comprehensive support for each supported style.
  - Don't try to force people."
- Unterstützung des Software-Designs:
  - "Support sound design notions.
  - Provide facilities for program organization.
  - Say what you mean.
  - All features must be affordable.

  - It is more important to allow a useful feature than to prevent every misuse.
  - Support composition of software from separately developed parts."

- sprachtechnische Regeln:
  - "No implicit violations of the static type system.
  - Provide as good support for user-defined types as for built-in types.
  - Locality is good.
  - Avoid order dependencies.
  - If in doubt, pick the variant of a feature that is easiest to teach.
  - Syntax matters (often in perverse ways).
  - Preprocessor usage should be eliminated."
- Unterstützung maschinennaher Programmierung:
  - "Use traditional (dumb) linkers.
  - No gratuitous incompatibilities with C.
  - Leave no room for a lower-level language below C++ (except assembler).
  - What you don't use, you don't pay for (zero-overhead rule).
  - If in doubt, provide means for manual control."

Vor diesem Hintergrund wurden konkrete Sprachdesignentscheidungen getroffen:

- als Verarbeitungsmodell dient das Konzept abstrakter Datentypen, nicht das Kommunikationsmodell,
- es wird zwischen Klassen und Instanzen unterschieden; Klassen sind gleichzeitig auch Module und regeln die Sichtbarkeit von Variablen und Methoden von außen (*public, protected, private*);
- multiple Vererbung zwischen Klassen; alle Instanz- und Klassenvariablen sowie alle Methoden einer Klasse werden an ihre Subklassen (*derived classes*) vererbt;
- Klassen sind in das Typkonzept integriert; es gibt primitive Typen außerhalb der Klassenhierarchie;
- statische Typisierung von temporären sowie von Instanz- und Klassenvariablen und von Funktionsergebnissen; Methodendispatch zur Compilezeit oder zur Laufzeit bei *virtual functions*;
- Templates und Exception Handling;
- keine Methodenkombination,
- keine Metaklassen und keine Metaobjektprotokolle, nur minimale Typinformationen zur Laufzeit: *Run-Time-Type-Info (RTTI).*

Stroustrup hat sich bewußt gegen Metaobjekte mit der Begründung entschieden [1994, S.324]:

> "... In essence, having a meta-object mechanism embeds an interpreter for the complete language in the run-time environment."

Dieses Vorurteil aus den Erfahrungen mit CLOS und Smalltalk, wird mit ΤΕΛΟΣ aber eindeutig widerlegt. Methodenkombination wurde verworfen, weil sie gegen die Regel der Vermeidung von Reihenfolgeabhängigkeiten verstößt und weil das Konzept von CLOS als zu komplex eingeschätzt wurde [Stroustrup, 1994, S. 268]. Da Vererbung nicht nur zur Spezialisierung und Generalisierung von Klassen verwendet werden darf und soll, kann es auch keine "natürliche" Linearisierung der Vererbungshierarchie geben. Daher kann es auch keine "nächstallgemeinere" Methode geben.

Insgesamt kann man zweifellos anerkennen, daß der Erfolg von C++ sich nicht zufällig eingestellt hat, sondern auf die richtige Designstrategie für die gewählten Ziele zurückzuführen ist. Wesentlich ist die realistische Einschätzung, mit welchen Problemen sich 80 % der betriebssystemnahen Software-Entwickler täglich herumschlagen. Insbesondere die Lisp-Gemeinde sollte von C++ gelernt haben, daß man die Lösung von Effizienzproblemen nicht auf die übernächste Rechnergeneration vertagen darf, will man nicht in die Nischen der Bedeutungslosigkeit geraten. Der technisch und kognitiv begründete Bedarf nach einer höheren problem- und anwendungsorientierten Sprache, im Unterschied zu maschinenorientierten, kann C++ jedoch nicht abdecken. Es bleibt also Raum für weitere Sprachentwürfe, zumal die heutigen PCs um Größenordnungen leistungsfähiger sind als beispielsweise die museumsreifen[12] Lispmaschinen.

3.3.3 Java

Knüpft man an das eben gesagte an, so ist Java [Gosling *et al.*, 1996] eher ein Beispiel für unvorhergesehenen Markterfolg. Java wurde zunächst für eingebettete Software-Systeme zur Steuerung von elektronischen Haushaltsgeräten entwickelt. Dabei muß die Software in kürzester Zeit auf neue Hardwarebausteine portiert werden. Ebenso entscheidend ist die hohe Zuverlässigkeit der Geräte und somit der Software. Ein kleines Team bei Sun Microsystems unter der Leitung von James Gosling kam schnell zu dem Schluß, daß existierende Programmiersprachen wie C und C++ diesen Anforderungen nicht standhalten. 1990 begann James Gosling daher, eine neue Programmiersprache zu entwickeln. Mit der Ausbreitung des World Wide Web im Internet wurde deutlich, daß dort sehr ähnliche Eigenschaften einer Programmiersprache gefragt sind. Die Einführung von Applets (kleiner Programme, die über das Netz transportiert werden können, um auf einem Client-Rechner ausgeführt zu werden) brachte Java schließlich den Durchbruch.

Java wird von Sun schlagwortartig mit den Eigenschaften

- einfach,
- objektorientiert,

[12] Es gibt tatsächlich eine lesenswerte Web-Seite, die sich Lispmaschinen-Museum nennt: http://home.brightware.com/~rwk/symbolics/

- verteilt,
- interpretiert,
- robust,
- sicher,
- architekturunabhängig,
- portabel,
- effizient,
- nebenläufig und
- dynamisch

charakterisiert. Java ist einfacher als C und C++, weil fehlerträchtige, unsichere und maschinennahe Sprachkonstrukte wie manuelle Speicherverwaltung, Pointer, Sprunganweisung, Prekompiler (Makros), multiple Vererbung und Überladen von Operatoren nicht in die Sprachdefinition aufgenommen wurden.

Im Unterschied zu C++ wurde Java von Grund auf objektorientiert entworfen. Es mußten keine Kompatibilitätskompromisse mit einer bereits existierenden Programmiersprache eingegangen werden. Aus didaktischen Gründen sollte Java nur syntaktisch weitgehend C und C++ ähneln. Objektorientierung wird grundsätzlich wie in C++ als Datenabstraktion mit Vererbung verstanden. Es wird zwischen Klassen und Instanzen unterschieden, Klassen übernehmen zudem die Rolle von Modulen. Alle Definitionen und Anweisungen befinden sich syntaktisch innerhalb von Klassendefinitionen. Es gibt spezielle abstrakte Klassen, genannt *interfaces*, die der Schnittstellenspezifikation dienen und multiple Vererbung zulassen. Spezifikationen kann man daher mehrfach wiederverwenden, Implementierungen aber nur einfach. Im Unterschied zu Smalltalk wird zwischen primitiven Datentypen und Klassen bzw. zwischen primitiven Werten und Objekten unterschieden. Primitiv sind *boolean, char, byte, short, int, long, float* und *double*. Diese werden nach der Wertsemantik behandelt, während alle (anderen) Objekte der Referenzsemantik folgen. Aus praktischer Sicht der Implementierer mag diese Mischung beider Prinzipien in einer Sprache als eine vertretbare Designentscheidung erscheinen. Sie erschwert aber beträchtlich das Programmieren.

Java ist ausdrücklich als *verteilte* Sprache entworfen worden, um die Entwicklung von Netzanwendungen zu unterstützen. Dies schließt Client-Server-Konzepte ein, direkte Rechnerkommunikation über *Sockets* und die Integration mit Web-Browsern. Eine Schlüsselrolle spielen dabei *Applets*. Sie können direkt in HTML-Dokumente eingebunden werden, um beim Betreten einer Web-Seite auf dem Client-Rechner ausgeführt zu werden. Auch die verteilte Programmentwicklung wird in Java dadurch unterstützt, daß Klassen in Bytecodeformat über das Netz dynamisch geladen werden können. Spezielle Vorkehrungen wurden getroffen, um die Kompatibilität und Integrität abhängiger Klassen bei Versionsänderungen ohne ihre Recompilation sicherzustellen.

Java wird, wie Smalltalk, insofern *interpretiert* als Quellcode zunächst in Bytecode übersetzt wird, der dann von einer virtuellen Maschine ausgeführt (interpretiert) wird. Dadurch, daß alle wichtigen Rechner- und Betriebssystemplattformen einen Interpretierer für Java-Bytecode bereitstellen, wird ein sehr hohes Maß an Portabilität erreicht. Insbesondere alle gängigen Web-Browser enthalten inzwischen einen Java-Interpretierer. Wie schon bei Smalltalk [Deutsch, 1989] übersetzen die neuesten Interpretierer *Just-In-Time (JIT)* den Bytecode zunächst in Maschinencode, der dann genauso effizient ausgeführt wird, wie C++ Binärcode. Diese Technik verbindet die Vorteile der Bytecode-Interpretierer mit der Effizienz optimierender Compiler.

Java ist *robust* und *sicher*, weil auf maschinennahe Sprachkonstrukte verzichtet wurde (s.o.), Typfehler weitgehend zur Übersetzungszeit gemeldet werden und mögliche Laufzeitfehler durch entsprechende Prüfungen wie Indexgrenzen von Arrays kontrolliert und signalisiert werden. Das *Exception-Konzept* trägt auch dazu bei, daß mögliche Fehler in Anwendungen abgefangen und behandelt werden können. Ferner wird der Bytecode vor seiner Ausführung auf mögliche Verletzungen von Zugriffsbeschränkungen überprüft. Insbesondere in Verbindung mit Applets wird dadurch Mißbrauch und Manipulation weitgehend ausgeschlossen.

Java unterstützt *Nebenläufigkeit* in Programmen durch *Threads* und *Locks*. Wie die virtuelle Javamaschine mehrere Threads ausführt, ob auf mehreren Hardware-Prozessoren oder auf einem Prozessor durch Time-Slicing bleibt offen. Dies erhöht ebenfalls die *Architekturunabhängigkeit* und *Portabilität*.

Die *Effizienz* von Java ließ zunächst viel zu wünschen übrig. Im Vergleich zu C und C++ war Java 50 bis 100 mal langsamer und benötigte das mehrfache an Hauptspeicher. Mit der Etablierung der Just-In-Time-Compiler (JIT) ist dieser Abstand für die meisten Anwendungen nahezu aufgeholt. Im Vergleich zu Smalltalk, oder auch Lisp, läßt sich Java wegen der statischen Typisierung effizienter realisieren. Gegenüber C++ trägt die Einfachheit zu prinzipiell besseren Lösungen bei.

Die *Dynamik* von Java ergibt sich aus seiner Objektorientierung, der Möglichkeit Klassen (in Bytecode-Format) zur Laufzeit bei Bedarf nachzuladen, Klassen als Objekte erster Ordnung zu behandeln und entsprechende Introspektions-Sprachmittel zu benutzen (*reflection package*). Im Vergleich zu Smalltalk ist die Dynamik durch die statische Typisierung von Variablen eingeschränkt. Im Vergleich zu Lisp und Dylan fehlt darüberhinaus die Möglichkeit der syntaktischen Erweiterung der Sprache. Erweiterbare Metaobjektprotokolle wie in CLOS und ΤΕΛΟΣ wurden in Java auch noch nicht aufgenommen.

Ein interessantes Konzept, das für eine bessere Interoperabilität von Softwarebausteinen auf Betriebssystemebene sorgt, stellen sogenannte *Java-Beans* dar. Sie können ebenso als Modularisierungsmittel angesehen werden.

Abgesehen von ΤΕΛΟΣ, entspricht Java insgesamt am ehesten den Designkriterien aus Kapitel 2. Ein detaillierter Vergleich mit ΤΕΛΟΣ erfolgt in Kapitel 8.4.

3.4 Objektsysteme für Lisp

Lisp ist neben Fortran und Cobol eine der ältesten noch verwendeten Programmiersprachen. Sie wurde Ende der 50er Jahre von John McCarthy am MIT für Forschungsexpe-

rimente der Künstlichen Intelligenz erfunden. Von Beginn an zeichnete sich Lisp durch vier charakteristische Eigenschaften aus: durch die Ausrichtung auf die symbolische Informationsverarbeitung, im Unterschied zur numerischen in Fortran, durch die theoretische Fundierung im Lambda-Kalkül [Barendregt, 1984] und dem daraus folgenden applikativen Programmierstil, im Unterschied zum prozeduralen in Fortran, sowie durch die Repräsentation von Programmen als Liste - der originären Lisp-Datenstruktur. Insbesondere diese letzte Eigenschaft löste einen Schub von innovativen Konzepten des interaktiven Programmierens mit Entwicklungsumgebungen, des Compilerbaus und nicht zuletzt von zahlreichen in Lisp eingebetteten Wissensrepräsentationssprachen aus. Gleichzeitig führte dies zu auseinanderlaufenden Sprachdialekten, was die kommerzielle Benutzung von Lisp erschwerte. Die wichtigsten Lisp-Dialekte sind heute CommonLisp [Steele Jr., 1990a] im kommerziellen und im Forschungsbereich sowie Scheme [IEEE Std 1178-1990, 1991] im Ausbildungsbereich.

Es ist bemerkenswert, daß Scheme und CommonLisp im Prinzip durchaus ähnlich sind, auf der anderen Seite gibt es aber Unterschiede wie sie größer kaum sein könnten: der ANSI Scheme Standard umfaßt rund 50 Seiten, ANSI Common Lisp hingegen rund 1500 Seiten! In diesem Zusammenhang stellen Steele und Gabriel fest [1996, S. 309]:

> ... you can put together a working Lisp interpreter in less than a day if you know what you are doing. ... And, if you want to put together a fairly decent, but not outstanding, Common Lisp system, you should plan on 20 to 30 man-years.

Angesichts dieser Extreme ist es vielleicht verständlich, wenn in Europa nach einem Mittelweg gesucht wurde [Padget *et al.*, 1986]. Das Ergebnis gemeinsamer Anstrengung in der EuLisp-Working-Group ist ein moderner Lisp-Dialekt EuLisp [Padget *et al.*, 1993], den Steele und Gabriel mit den Worten kommentieren [1996, S. 257]:

> The definition of EuLisp took a long time ... Nevertheless, the layered definition, the module system and the object-orientedness from the start demonstrate that new lessons can be learned in the Lisp world.

Als kommerzielle Lisp-Implementierung, die wesentliche Konzepte von EuLisp aufgegriffen hat, ist IlogTalk [IlogTalk, 1994] als Weiterentwicklung des Lisp-Dialekts Le-Lisp [Chailloux *et al.*, 1991], [LeLisp, 1992] zu nennen. Es darf auch erwähnt werden, daß die von Apple entwickelte dynamische objektorientierte Programmiersprache dylan [Shalit, 1996] gleiche Ziele und Designprinzipien aufweist wie EuLisp. Aus Marketinggründen vermeiden aber Apple und Digitool[13] jegliche Hinweise auf die (Eu)Lisp-Herkunft von dylan.

Ich verzichte an dieser Stelle bewußt auf eine weitere Darstellung von Lisp insgesamt und beschränke mich auf entsprechende Literaturhinweise.[14] Eine Einführung in Lisp mit einer kurzen Geschichtsdarstellung, den Grundelementen der Sprache und dem Schwerpunkt auf objektorientierter Programmierung in Lisp ist in [Christaller, 1988] zu finden.

[13] Wegen wirtschaftlicher Probleme hat Apple das dylan-Entwicklungsteam aufgelöst. Die von ehemaligen Mitarbeitern dieses Teams neu gegründete Firma Digitool hat dylan übernommen.

[14] Bei der Diskussion der Implementierungstechniken in Kapitel 5 werden die wichtigsten Sprachkonstrukte eingeführt.

Ausführlich wird die Geschichte von Lisp in [Stoyan, 1980], [McCarthy, 1981] und [Steele und Gabriel, 1996] behandelt. Techniken des Interpreter- und Compilerbaus für Lisp aus den 70ern findet man in [Allen, 1978]. Ausführlich und theoretisch sehr fundiert diskutiert Queinnec [1996] alle wichtigen Sprachkonzepte von Lisp aus heutiger Sicht. Ein Klassiker der Einführung in allgemeine Konzepte der Informatik auf der Grundlage von Scheme ist [Abelson und Sussman, 1985]. Ebenso empfehlenswert zur funktionalen Programmierung in Scheme ist [Springer und Friedman, 1989] sowie [Friedman *et al.*, 1992] für Interpretierer-Konstruktion und Continuation-Passing-Style (CPS). Interessant ist auch die Einschätzung verschiedener Entwicklungstendenzen um Lisp aus der Sicht Anfang der 90er [Gabriel, 1993] und ein gegenwärtiger Überblick in [Laddaga und Veitch, 1997].

Nun komme ich zu den objektorientierten Erweiterungen von Lisp, die eine lange Tradition haben, weil Lisp besonders einfach erweiterbar ist. Die größte Verbreitung haben hier die Systeme LOOPS [Bobrow und Stefik, 1981] und FLAVORS [Weinreb und Moon, 1981] erreicht, die schließlich zur Definition von CLOS [Bobrow *et al.*, 1988] geführt haben. Das innovative an CLOS im Vergleich zu FLAVORS ist die bessere Integration objektorientierter Sprachmittel mit dem funktionalen Kern von COMMONLISP, sowie sein Spracherweiterungsprotokoll, auch Metaobjektprotokoll genannt.

In der folgenden Darstellung beschränke ich mich auf eine Teilsprache von FLAVORS, nämlich das Micro Flavor System (MFS) [Christaller, 1988], den reflexiven Kern von ObjVlisp [Cointe, 1987] sowie auf das Common Lisp Object System (CLOS). Als weitere Objektsysteme für Lisp und Scheme seien kurz erwähnt:

- Poor Mans Flavor System (PMFS) [di Primio und Christaller, 1983],
- Micro Common Flavors (MCF) [di Primio, 1988],
- MEROON [Queinnec, 1993], MEROONET [Queinnec, 1996],
- OakLisp [Lang und Pearlmutter, 1986], [Lang und Pearlmutter, 1988].

Im Hinblick auf die neuesten Entwicklungen ist dabei MEROON besonders hervorzuheben. Einen Überblick zu Objektsystemen für Scheme findet man in der von mir betreuten Diplomarbeit [Lange, 1993]. Natürlich müssen an dieser Stelle auch die Systeme MCS und CELOS genannt werden, die in den folgenden Kapiteln als Resultat dieser Arbeit vorgestellt werden.

3.4.1 MFS: Micro Flavor System

Das Micro Flavor System [Christaller, 1988] ist eine sogenannte Mikroversion des für die MIT-Lispmaschinen entwickelten FLAVORS [Weinreb und Moon, 1981]. Zuvor wurde schon das Poor Mans Flavor System (PMFS) als ein portables Objektsystem für Lisp realisiert [di Primio und Christaller, 1983]. Der Entwurf und die Implementierung von MFS hatte primär didaktische Ziele und eignet sich daher besonders gut, um die Kerngedanken von FLAVORS hier darzustellen. Ebenso gut kann an MFS die Implementierungstechnik der Spracheinbettung gezeigt werden.

Dem MFS liegt wie Smalltalk das Kommunikationsmodell zu Grunde. Objekte führen Berechnungen durch, indem sie Nachrichten austauschen und verarbeiten. Man unterscheidet

auch zwischen *generischen Objekten*, die *Flavors* genannt werden, und ihren Konkretisierungen, die *Instanzen* genannt werden. Es handelt sich also um ein zweistufiges Objektsystem. Die generischen Objekte sind jedoch wie in FLAVORS und C++ keine Objekte erster Ordnung. Sie werden aber ähnlich wie Instanzen implementiert, so daß das MFS entsprechend leicht erweitert werden kann.

Der Austausch von Nachrichten zwischen Objekten geschieht über ein spezielles Sprachkonstrukt `send`:[15]

`(SEND` *instanz selektor argument*$_1$ `...)`

Es drückt ein *Sendeereignis* aus, wobei an das Objekt *instanz* die Nachricht mit dem Inhalt: *selektor argument*$_1$ ... gesendet wird. Der *selektor* bestimmt, welche Methode ausgewählt und auf *argument*$_1$ ... angewandt wird. In Methoden ist das Empfängerobjekt implizit über die Variable `self` sichtbar. `SEND` kann hier als Funktion angesehen werden, die auf die Argumente *instanz, selektor, argument*$_1$ ... angewandt wird. An jeder Argumentstelle kann ein beliebiger Ausdruck stehen, auch ein Sendeereignissen. Bei geschachtelten Sendeereignissen spricht man auch von *fortgesetztem* (oder *kaskadiertem) Nachrichtenversenden.*

Instanzen werden über den Konstruktor `create-instance` erzeugt:

`(CREATE-INSTANCE` *flavor instanzvariable*$_1$ *aktueller-wert*$_1$ `...)`

Dabei können optional die für *flavor* definierten Instanzvariablen (einschließlich der geerbten) aktuelle Initialisierungswerte erhalten.[16] Bezogen auf die Programmausführung, muß zum Zeitpunkt der Instanzerzeugung das entsprechende Flavor bereits definiert worden sein.[17] Eine Flavor-Definition bestimmt die Struktur und das Verhalten seiner Instanzen.

`(DEFCLASS` *flavor* `(`*instanzvariable*$_1$ `...) (`*superklasse*$_1$ `...))`

Dabei werden die Instanzvariablen textuell innerhalb einer Flavor-Definition angegeben. Im MFS werden für jede Instanzvariable automatisch entsprechende Lese- und Schreibmethoden erzeugt, Instanzvariablen können nur über entsprechende Sendeereignisse gelesen bzw. geschrieben werden.[18]

Weitere Methodendefinitionen erfolgen textuell außerhalb der Flavor-Definition:

`(DEFACTION (`*flavor selektor*`) (`*parameter*$_1$ `...)` *methodenrumpf*`)`

[15] Hier benutze ich bereits die sehr einfache Lisp-Syntax. Alle zusammengesetzten Ausdrücke werden geklammert, es gilt die Präfixnotation von Operationen, d. h. die erste Stelle nach einer öffnenden Klammer wird als Anwendung eines vordefinierten bzw. benutzerdefinierten syntaktischen Konstrukts (Konditional, Konstruktor) oder einer vordefinierten bzw. benutzerdefinierten Funktion interpretiert. In der Regel, insbesondere bei Funktionsanwendungen, werden alle folgenden Stellen zunächst ausgewerten und dann als aktuelle Argumente an die formalen Parameter der anzuwendenden Funktion gebunden. Syntaktische Konstrukte (auch benutzerdefinierte Makros) regeln die Interpretation der folgenden Stellen selbst.

[16] Streng genommen gilt für die Mikroimplementierung, daß hier auch andere Name-Wert-Paare vorkommen dürfen, sie haben aber keine Auswirkung auf den von außen beobachtbaren Zustand einer Instanz.

[17] Für die Mikroimplementierung gilt diese Einschränkung nicht. Hier genügt es, wenn das Flavor vor dem ersten Nachrichtenereignis für eine seiner Instanzen definiert wird.

[18] In FLAVORS können Instanzvariablen der Empfänger-Instanz innerhalb eigener Methoden wie freie Variable referiert werden. Welche von ihnen entsprechende Lese- und Schreibmethoden erhalten, um auch von außen benutzbar zu werden, kann bei der Flavor-Definition spezifisch gesteuert werden.

Damit wird diese neue Methode im sogenannten *Protokoll* (Methodenverzeichnis) des *flavor* mit dem *selektor* assoziiert. Folgendes kleines Beispiel soll die Verwendung der eingeführten Sprachkonstrukte verdeutlichen:

```
(defclass pair       ; name des flavor
   (left right)      ; liste der instanzvariablen
   ())               ; liste der superklassen

(defaction (pair first) ()
   (send self 'left))

(setf pair1 (create-instance 'pair 'left 0))

(send pair1 'first)  ; ==> 0
```

Das MFS untertützt wie FLAVORS die *allgemeine multiple Vererbung*, d. h. ein Flavor kann von einem oder mehreren Flavors Informationen erben, wenn in der Liste der Superklassen entsprechende Flavors spezifiziert werden. Die Superklassenbeziehung induziert einen gerichteten azyklischen *Flavor-Graphen*. Für ein gegebenes Flavor wird die transitive Hülle der direkten Superklassenbeziehung auch als seine *Komponenten* bezeichnet. Um die Mehrdeutigkeiten bzgl. der Methodenauswahl und bzgl. der *Ersatzwerte für Instanzvariablen*[19] aufzulösen, werden die Komponenten linearisiert. Dabei wird der Algorithmus *Depth-First-Left-to-Right* verwendet. Leider liefert dieser Algorithmus für einige ausgefallene Vererbungshierarchien keine intuitiv gewünschten Komponentenlisten [Britthauer *et al.*, 1989a].

Da es sich bei MFS um eine Mikroversion handelt, wurde auf folgende Sprachkonstrukte aus dem großen Flavor-System verzichtet: Methodenkombination, Ersatzwerte für Instanzvariablen etc. Die didaktisch besonders gelungene und extrem kurze Implementierung von MFS auf nur zweieinhalb Seiten kommentierten Lisp-Codes [Christaller, 1988, S.31-33] enthält aber entsprechende Ankerstellen für Erweiterungen. Darauf werde ich in Kapitel 6 und 7 ausführlich eingehen.

3.4.2 ObjVLisp

Die Nachteile der unterschiedlichen Behandlung von Flavors und ihrer Instanzen wurden mit dem Hinweis auf eine bessere Lösung mit Hilfe von *Metaklassen* bereits in [Christaller, 1988, S.17] genannt. In OBJTALK [Laubsch, 1982], [Rathke und Laubsch, 1983] wurde ein minimaler Kern eines zweistufigen Objektsystems für Wissensrepräsentationszwecke vorgestellt. Im Prinzip die gleiche Lösung schlägt [Cointe, 1987], [Cointe, 1988] als reflektive Lisp-Architektur vor, um ein uniformes Objektsystem einer Programmiersprache zu definieren. Cointe kritisiert das Metaklassenkonzept von Smalltalk, weil Klassen und (terminale) Instanzen dort nicht uniform behandelt werden. Klassen seien zwar Objekte erster Ordnung und somit selbst (nicht-terminale) Instanzen einer (Meta-)Klasse, jedoch werden sie nicht uniform erzeugt und initialisiert. Metaklassen können in Smalltalk explizit gar

[19] Nur in FLAVORS, nicht in MFS.

nicht erzeugt werden, sie entstehen implizit als Folge einer Klassendefinition. Zwei Klassen sind daher auch nie Instanzen ein und derselben Metaklasse, d. h. jede Metaklasse hat genau eine Instanz. Hier wird es wirklich fragwürdig, ob man überhaupt noch von Klassen sprechen sollte.

Cointe gründet sein Modell auf sechs Postulaten [Cointe, 1988, S. 159-160]:

1. Ein Objekt besteht aus Daten und Prozeduren.
2. Objekte werden nur über Sendeereignisse (*message passing*) aktiviert.
3. Jedes Objekt gehört einer Klasse an, die seine Daten (Instanzvariablen) und sein Verhalten (Methoden) spezifiziert.
4. Jede Klasse ist auch ein Objekt, instanziiert durch eine andere Klasse, seine Metaklasse genannt. Die Ur-Metaklasse ist die Klasse `class`, die Instanz von sich selbst ist.
5. Eine Klasse kann als Subklasse einer (oder mehrerer) anderen Klasse(n) definiert werden, was *Vererbung* von Instanzvariablen und Methoden ermöglicht. Die Klasse `object` legt das Verhalten aller Objekte fest, sie ist die Wurzel der Vererbungshierarchie.
6. Klassenvariablen eines Objekts sind Instanzvariablen seiner Klasse (als Objekt).

Abbildung 3.1 verdeutlicht den reflektiven Kern von OBJTALK und ObjVlisp bestehend aus `object` und `class` sowie seine potentielle Beziehungen zu benutzerdefinierten Klassen und Instanzen (hier der Metaklasse `person-class`, der Klassen `window`, `person` und der terminalen Instanzen `Joe` und `Mary`).

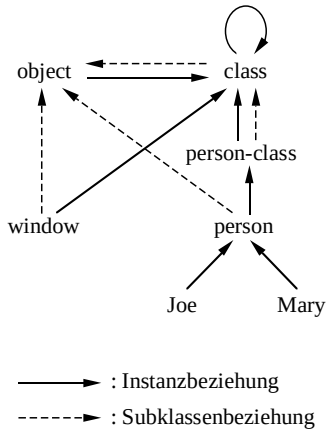


Abbildung 3.1: Reflektiver Kern in OBJTALK und ObjVlisp.

Der reflektive Kern von ObjVlisp besteht im wesentlichen aus zwei systemdefinierten Klassen mit einigen wenigen Methoden [Cointe, 1987]. Für `object` sind es die Methoden:

- `class`, um die Klasse des Objekts festzustellen,

- `initialize`, um ein Objekt zu initialisieren.
- `?` und `?<-`, um beliebige Instanzvariablen zu lesen und zu schreiben.

Und für `class` sind es die Methoden:

- `new`, um neue Objekte zu erzeugen,
- `basicnew`, um neue Objekte zu allozieren,
- `initialize`, um Klassen zu initialisieren und dabei den statischen Anteil der Vererbung durchzuführen, d. h. die Vereinigung der für die zu initialisierende Klasse direkt spezifizierten Instanzvariablen mit den Instanzvariablen ihrer Superklassen.
- `name`, `supers`, `i_v`, `methods`, um Klassen zu inspizieren,
- `understand`, um neue Methoden hinzuzufügen.

Im übrigen ist ObjVlisp eingebettet in LeLisp und enthält zusätzlich das als Funktion realisierte Konstrukt `send` zum Nachrichtenversenden. Methoden sind keine Objekte erster Ordnung, können also auch keine Nachrichten empfangen. Diese Einschränkung ist aber nicht prinzipiell notwendig. Sie vereinfacht jedoch die Implementierung und das Verstehen.

3.4.3 CLOS: Common Lisp Object System

CommonLisp [Steele Jr., 1984] ist Anfang der 80er Jahre im wesentlichen aus den Dialekten ZetaLisp, MacLisp und Interlisp[20] entstanden. Zunächst enthielt CommonLisp keine objektorientierten Sprachkonstrukte,[21] da Flavors aus ZetaLisp und loops aus Interlisp zu unterschiedlich waren. Wärend der ANSI-Standardisierung (Gruppe X3J13) in der zweiten Hälfte der 80er Jahre [Steele Jr., 1990b] wurde CommonLisp doch um ein weitgehend integriertes Objektsystem CLOS erweitert. CLOS stellt nach [Steele und Gabriel, 1996, S. 266] die wichtigste technische Entwicklung der letzten Zeit dar. 1986 konkurrierten vier Gruppen um den besten Entwurf eines Objektsystems für CommonLisp: NewFlavors (Symbolics) [Symbolics Inc, 1985], CommonLoops (Xerox) [Bobrow *et al.*, 1986], Object Lisp (LMI) [Drescher, 1987] und Common Objects (HP) [Kempf und Stelzner, 1987]. Schließlich wählte man NewFlavors und CommonLoops als Basis für ein neues Objektsystem, das von einem siebenköpfigen Team

> ... of the most talented members of X3J13 ...

[Gabriel, 1993, S. 32] spezifiziert werden sollte. Die beiden ersten Teile des Entwurfs [Bobrow *et al.*, 1988] wurden nahezu unverändert für den ANSI-Standard übernommen. Das Metaobjektprotokoll (MOP) [Kiczales *et al.*, 1991] für CLOS als dritter Teil des Entwurfs blieb im ANSI-Standard bisher unberücksichtigt. Dennoch stellen einige CommonLisp-Anbieter es zur Verfügung. Darüberhinaus kann die portable Implementierung PCL

[20] Diese Reihenfolge drückt die Kompatibilitätspriorität aus.

[21] Bis auf `defstruct`, das Typerweiterungen mit einfacher Vererbung unterstützt, aber keine Methoden bzw. generische Funktionen und auch keine Spezialisierung von Slots.

[Kiczales *et al.*, 1991] verwendet werden. Als einführende Literatur zu CLOS sind [DeMichiel, 1993a] und [Keene, 1989] zu nennen. Vergleiche mit anderen objektorientierten Sprachen findet man in [DeMichiel, 1993b] (mit C++), [Schmidt und Omohundro, 1993] (mit Eiffel und Sather) sowie [Cointe, 1993] (mit Smalltalk). Im Vergleich von [Kristensen, 1996] wurde auch CLOS miteinbezogen.

Während des Designs von CLOS [Bobrow *et al.*, 1993] stellten sich drei wichtige Aufgaben [Kiczales *et al.*, 1991, S. 3-6]:

- Es sollte eine weitgehende *Kompatibilität* bestehender unterschiedlicher Objektsysteme erreicht werden, weil es bereits umfangreiche Anwendungen gab.
- Die *Erweiterbarkeit* sollte sichergestellt werden, weil die Objektsysteme auch als Grundlage für spezielle Wissensrepräsentationssprachen verwendet wurden und man nie alle potentiellen Anforderungen im voraus erfassen kann.
- *Effizienz* sollte für ein breites Spektrum von Anwendungen erreicht werden. Dies ist nicht durch eine einzige Optimierungsstrategie einer Implementierung zu erreichen.

Um diese Probleme zu lösen, wurde ein neuer Weg des Designs und der Implementierung beschritten: für die eigentliche objektorientierte Programmiersparche CLOS hat man ein Implementierungsmodell (CLOS MOP) mit den Sprachmitteln von CLOS selbst spezifiziert. Nimmt man beides zusammen (CLOS und CLOS MOP), so erhält man ein reflektives, spezialisierbares Objektsystem, das nach außen und nach innen erweitert und wiederverwendet werden kann. Mit dem Gelingen dieses Ansatzes, so [Kiczales *et al.*, 1991, S.], könne gezeigt werden, daß entgegen der landläufigen Meinung in problemorientierten Sprachen effizientere Programme als in maschinenorientierten Sprachen geschrieben werden können.

Im folgenden werde ich nur noch von CLOS sprechen und nur bei Bedarf explizit zwischen der Objekt- und der Metaobjektebene differenzieren, zumal die Grenze in CommonLisp insgesamt ohnehin nicht konsequent gezogen wird.

Klassen und Instanzen

Im Prinzip enthält CLOS ähnliche Sprachkonstrukte wie MFS und den gleichen Klassenkern wie ObjVlisp. Folgendes fortlaufendes Beispiel soll dies verdeutlichen.

```
(defclass point ()
   ((x :initarg :x
       :initform 0
       :accessor point-x)
    (y :initarg :y
       :initform 0
       :accessor point-y)))

(defvar point1 (make-instance 'point :x 5))

(point-x point1)                              ; ==> 5
```

```
(point-y point1)                              ; ==> 0
(setf (point-y point1) 9)
(point-y point1)                              ; ==> 9
```

Zunächst wird eine Klasse `point` als Subklasse der Default-Superklasse `standard-object` spezifiziert (da die Liste der direkten Superklassen leer ist). Es werden für sie zwei *Slots* (Instanzvariablen) `x` und `y` mit den *Slot-Optionen* `:initarg`, `:initform` und `:accessor` spezifiziert. Die Slotoption `:initarg` und `:initform` bestimmen das Initialisierungsverhalten, wenn eine Instanz der Klasse `point` mit Hilfe des generellen Konstruktors `make-instance` erzeugt wird. Dabei können die Schlüsselwörter `:x` und `:y` verwendet werden, um konkrete Initialisierungsargumente anzugeben. Anderenfalls wird der unter `:initform` spezifizierte Ersatzwert-Ausdruck ausgewertet. `point1` erhält daher den Wert 5 für den Slot `x` und den Wert 0 für den Slot `y`. Die Slotoption `:accessor` bewirkt, daß generische Lese- und Schreiboperationen bzw. entsprechende Methoden für den betroffenen Slot erzeugt werden.

Vergleicht man andere Sprachen mit CLOS, so fallen deren deutliche Nachteile gegenüber CLOS auf: Da es in Java keine optionalen Parameter gibt, müssen dort meistens mehrere Konstruktoren definiert werden. In Smalltalk gibt es keine Slotoptionen, es müssen für alle Instanzvariablen explizite Zugriffsmethoden erzeugt werden und ihre Initialisierung muß über benutzerdefinierte Initialisierungsmethoden erfolgen.

Vererbung

CLOS unterstützt *multiple Vererbung* der Struktur (Slots) und des Verhaltens (Methoden) von Objektklassen. Potentielle Mehrdeutigkeiten werden durch eine topologische Sortierung der Klassenhierarchie (alle direkten und indirekten Superklassen) aufgelöst. Für jede Klasse wird die sogenannte *class precedence list* berechnet. Diese enthält die Klasse selbst, gefolgt von der nächstallgemeineren Superklasse usw. einschließlich der Klasse `T`, der Superklasse aller Klassen. Dabei gilt, daß die Ordnung der direkten Superklassenbeziehungen nicht verletzt werden darf und daß bei mehreren direkten Superklassen die spezielleren links stehen. Allerdings werde ich im nächsten Kapitel zeigen, daß der CLOS-Algorithmus nicht immer ein intuitiv gewünschtes Resultat liefert.

Wie Slotoptionen vererbt werden zeigt folgende Fortsetzung des obigen Beispiels.

```
(defclass colored-mixin ()
   ((color :initform 'black
           :accessor color-of)))

(defclass blue-point (colored-mixin point)
   ((color :initform 'blue)))

(defvar point2 (make-instance 'blue-point))
(color-of point2)                             ; ==> BLUE
```

Der Farbaspekt von Objekten wird generalisiert und in einer sogenannte Mixin-Klasse `colored-mixin` definiert. Sie enthält einen Slot `color` mit dem Ersatzwert `black` und

dem Accessor `color-of`. In der Subklasse `blue-point` wird der Ersatzwert spezialisiert.[22] Instanzen von `blue-point` erhalten für den Slot `color` also den Initialwert `blue`, und zwar als Auswertungsergebnis des Ausdrucks `'blue`. Wichtig ist hierbei, daß der Slot `color` nur einmal in Instanzen von `blue-point` enthalten ist und nicht zweimal wie in C++ oder Java. Somit unterstützt nur CLOS eine spezialisierende Vererbung von Slotoptionen. Dabei gilt, daß

- die `:initarg`-Werte vereinigt werden,
- der speziellste `:initform`-Wert die allgemeineren überdeckt,
- die `:accessor`-, `:reader`-, und `:writer`-Werte vereinigt werden,
- der speziellste `:allocation`-Wert die allgemeineren überdeckt,
- alle `:type`-Werte mit der logischen Operation `and` verknüpft werden.

Generische Funktionen und Methoden

Ein wesentlicher Unterschied, auch zu Smalltalk, ist der Verzicht auf die Metapher des Nachrichtenversendens, und somit auf das Konstrukt `send`. Die Begriffe Sendeereignis und Funktionsanwendung verschmelzen. Statt

`(send` *empfänger selektor argument*$_1$ `...)`

schreibt man:

`(`*selektor empfänger argument*$_1$ `...)`

Auf diese Weise bekommen die Selektoren (Methoden-Namen) eine objekt- bzw. klassenübergreifende Bedeutung. Man kommt schließlich zum Konzept der generischen Funktionen. Dabei verliert der *empfänger* seinen besonderen Status und wird zum (ersten) Argument. Gleichzeitig bekommen die Argumente $argument_1, \ldots$ potentiell auch die Möglichkeit, die Methodenauswahl mitzubestimmen. Wenn nun einige hierin eine Abkehr von der Objektorientierung bzw. einen Rückschritt zur Prozedurorientierung sehen, so ist dies m. E. eine Fehleinschätzung. Warum soll der Fokus auf nur ein Objekt erzwungen werden, wo es doch genug Problemstellungen gibt, bei denen zwei oder mehr Objekte gleichberechtigt zusammenwirken? Die Probleme der Kritiker rühren wohl auch aus den Beschränkungen in Smalltalk, C++ und Java, daß eine Methodendefinition sich innerhalb genau einer Klasse befinden muß. Dies hängt wiederum mit der Doppelrolle von Klassen als Typ und als Modul in diesen Sprachen zusammen. In CLOS haben Klassen nur die Rolle eines Typs, Generische Funktionen und Methoden werden textuell außerhalb von Klassen definiert. Es ist daher durchaus natürlich, Multimethoden einzuführen. Sie bieten eine bessere Integration objektorientierter und funktionaler bzw. applikativer Sprachkonstrukte.

Eine generische Funktion faßt Methoden mit dem gleichen Namen und kongruenten Parametern zusammen.[23] In CLOS gibt es daher die Kontrukte `defmethod` und `defgeneric`.

[22] Man sagt auch *überschrieben*, was aber eine seiteneffektorientierte Vorstellung von Vererbung suggeriert.

[23] Man könnte auch von Signaturen sprechen, nur wird in CLOS der Ergebnistyp einer generischen Funktion bzw. einer Methode nicht spezifiziert.

Letzteres ist optional, d. h. daß die erste Auswertung eines `defmethod` Ausdrucks implizit eine neue generische Funktion erzeugt, falls sie noch nicht existiert.

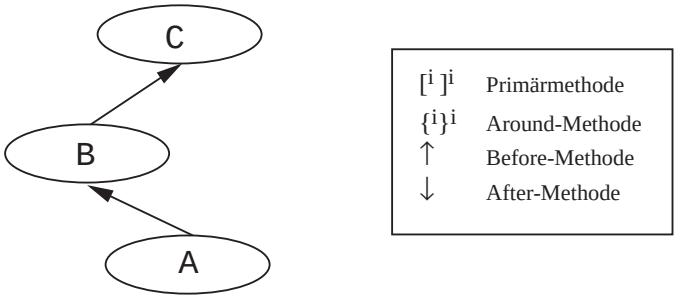
Hier ist ein Beispiel für Multimethoden, wobei die Verbindung zwischen zwei Punkten angezeigt wird:

```
(defgeneric display-connection (object1 object2))
(defmethod display-connection ((p1 point) (p2 point))
   (print "point<=>point"))
(defmethod display-connection ((p1 blue-point) (p2 point))
   (print "first-is-blue")
   (call-next-method))
(defmethod display-connection ((p1 point) (p2 blue-point))
   (print "second-is-blue")
   (call-next-method))
(defmethod display-connection ((p1 blue-point) (p2 blue-point))
   (print "both-are-blue")
   (call-next-method))

(display-connection point1 point1)
; ==> "point<=>point"
(display-connection point2 point1)
; ==> "first-is-blue"
;     "point<=>point"
(display-connection point1 point2)
; ==> "second-is-blue"
;     "point<=>point"
(display-connection point2 point2)
; ==> "both-are-blue"
;     "first-is-blue"
;     "second-is-blue"
;     "point<=>point"
```

Methodenkombination

Die Auswahl der anwendbaren Methoden für einen gegebenen Aufruf einer generischen Funktion und die Reihenfolge ihrer Anwendung richtet sich nach den Superklassenlisten der Klassen der aktuellen Argumente, nach der Art der Methodenkombination und nach der Argumentreihenfolge der generischen Funktion. Den einfachsten Fall haben wir im obigen Beispiel: es handelt sich um die Standardmethodenkombination, es gibt nur Primärmethoden und die Argumentreihenfolge geht von links nach rechts. Dabei wird jeweils die speziellste Methode zuerst ausgeführt, dann ausgelöst durch den expliziten Aufruf von `call-next-method` die nächstallgemeinere Methode usw. Die Standardmethodenkombination läßt neben Primärmethoden auch mit `:before`, `:after` oder `:around` qualifizierte Methoden zu, siehe Seite 59.



$\{^1_A\ \{^2_B\ \{^3_C\ \uparrow_A \uparrow_B \uparrow_C [^1_A [^2_B [^3_C\ ]^3_C]^2_B]^1_A \downarrow_C \downarrow_B \downarrow_A\ \}^3_C\ \}^2_B\ \}^1_A$

Abbildung 3.2: Methodenkombination in CLOS.

Abbildung 3.2 zeigt am Beispiel von drei Klassen `A, B` und `C`, wobei `A` Subklasse von `B` und `B` Subklasse von `C` ist, und einer generischen Funktion `f(x)`, in welcher Reihenfolge die Methoden für eine Instanz der Klasse `A` ausgeführt werden müssen. Dabei wird unterstellt, daß für jede Klasse alle vier Methodenarten definiert wurden und daß in allen Around-Methoden und in zwei Primärmethoden (für die Klassen `A` und `B`) die jeweilige nächst allgemeinere Methode aufgerufen wird. Der allgemeine Mechanismus der Standardmethodenkombination von CLOS wird in Kapitel 6.2.2 auf Seite 157 genau beschrieben.

Mit Hilfe der Methodenkombination kann man abstrakter und deklarativer die Rolle von Methoden ausdrücken. Allerdings kann sie im Zusammenhang mit der in CLOS bereitgestellten allgemeinen multiplen Vererbung und der Möglichkeit, gleich mehrere Methoden für eine Klasse zu definieren, auch zu undurchschaubaren Programmen führen [Bretthauer *et al.*, 1989c]. In Kapitel 8.3.4 wird das Konzept der Methodenkombination erneut aufgegriffen.

Über die Standardmethodenkombination hinaus unterstützt CLOS weitere vordefinierte (`+, and, append, list, max, min, nconc, or` und) sowie benutzerdefinierte Methodenkombinationen mittels `define-method-combination`.

Klassen- vs. instanzspezifische Methoden

Methoden sind in CLOS in der Regel klassenspezifisch, d. h. für alle direkten Instanzen einer Klasse gelten die gleichen Methoden. Es gibt aber auch instanzspezifische Methoden, die nur dann anwendbar sind, wenn das entsprechende aktuelle Argument und die spezifizierte Instanz, der sogenannte *Eql-Specializer*, identisch bzw. gleich sind.

So kann beispielsweise eine Eql-Methode `color-of` speziell der Zahl 7 zugeordnet werden:

```
(defmethod color-of ((object (eql 7))) 'seven-color)

(color-of 7)                              ; ==> SEVEN-COLOR
```

Die textuelle Trennung von Klassen- und Methodendefinitionen erlaubt es, neue Protokolle für bereits definierte Klassen zu definieren ohne den möglicherweise gar nicht zugänglichen Quellcode zu ändern. Beispielsweise kann das Protokoll aller Objekte um den Farbaspekt durch eine einfache Methodendefinition erweitert werden:

```
(defmethod color-of (object) 'default-color)

(color-of point1)                              ; ==> DEFAULT-COLOR
```

Lese- und Schreiboperationen

Die Tatsache, daß Lese- und Schreiboperationen generisch sind, und die Standardmethodenkombination unterstützen, kann dazu genutzt werden, z. B. vor dem Setzen eines Slots den neuen Wert auf seine Zulässigkeit zu überprüfen. Dies läßt sich am besten als *Before-Methode* realisieren:

```
(defmethod (setf color-of) :before (new-color ((p blue-point))
   (unless (possible-color-p new-color)
      (error "Setting wrong color ~S for ~S."
             new-color
             p)))

(defun possible-color-p (color) (eq color 'blue))

(setf (color-of point2) 'black)             ; ==> ERROR
```

Zusätzlich zu den generischen Lese- und Schreiboperationen für spezifische Slots gibt es ein (primitiveres) allgemeines Konstrukt `slot-value`, das den Zugriff auf beliebige Slots erlaubt wie das folgende Beispiel zeigt:

```
(defmethod print-object ((p point) stream)
   (format stream
           "#<~S ~S ~S>"
           (class-name (class-of p))
           (slot-value p 'x)
           (slot-value p 'y)))

point1                                      ; ==> #<POINT 0 0>
point2                                      ; ==> #<BLUE-POINT 0 0>
```

Im Methodenrumpf von `print-object` tauchen zwei weitere CLOS-Operationen auf: `class-of` und `class-name`. Im Prinzip gehören beide in die Metaobjektebene, sie stehen aber in CLOS schon auf der Objektebene zur Verfügung. Die Verwendung von `slot-value` ist nicht auf Methoden beschränkt. Die Semantik und Implementierung von generischen Zugriffsoperationen wird über `slot-value` spezifiziert, d. h. `color-of` besitzt eine Methode, die folgender Definition entspricht:

```
(defmethod color-of ((object colored-mixin))
   (slot-value object 'color))
```

`slot-value` selbst ist eine komplexere Operation, die ihrerseits die MOP-Operation `slot-value-using-class` (siehe Seite 66) aufruft. Aus Effizienzgründen darf und muß eine CLOS-Implementierung die Standardslotzugriffe daher optimieren.

Erzeugung und Initialisierung von Objekten

Objekte, die in der Regel terminale Instanzen sind, aber auch Klassenobjekte sein können, werden in CLOS mittels `make-instance` erzeugt, das

1. die Zulässigkeit der angegebenen Initialisierungsargumente prüft und ggf. Default-Initialisierungsargumente hinzufügt,
2. `allocate-instance` aufruft, um ein uninitialisiertes neues Objekt zu allozieren,

3. `initialize-instance` aufruft, das das neue Objekt mittels `shared-initialize` und/oder benutzerdefinierter Initialisierungsmethoden initialisiert, und

4. schließlich das neue Objekt als Ergebnis zurückliefert.

Alle eben genannten generischen Funktionen können vom Benutzer spezialisiert werden, d. h., es können für benutzerdefinierte Klassen eigene Methoden definiert werden, die mittels `call-next-method` auf systemdefinierte Standardmethoden zurückgreifen können. Dies ist ein sehr mächtiger und flexibler Mechanismus im Vergleich zu anderen Sprachen und Objektsystemen. Nachteil ist aber, daß man wegen vieler Möglichkeiten, dasselbe zu bewirken, leicht den Durchblick verlieren kann. Hierzu ein kleines Beispiel:

```
(defclass point ()
   ((x :initarg :x
       :initarg :y
       :initform 1
       :accessor point-x)
    (y :initarg :y
       :initarg :x
       :initform 2
       :accessor point-y)
        (z :initform 3
           :accessor point-z))
       (:default-initargs :x 11 :y 22))

(defmethod initialize-instance ((p point) &key (x 111) (y 222))
   (call-next-method))

(defvar point11 (make-instance 'point))

(point-x point11)                              ; ==> ?
(point-y point11)                              ; ==> ?
```

Beide Slots von `point1` `x` und `y` erhalten den Initialwert `11`.[24] Offensichtlich hat man beim Design des Initialisierungsprotokolls nicht ausreichen für die Orthogonalität der Sprachkonstrukte gesorgt.

Redefinieren von Klassen

Bei den letzten Programmzeilen könnte die Frage aufgekommen sein, ob es sich noch um das fortlaufende Beispiel handelt. Schließlich wurde die Klasse `point` bereits früher definier. In der Tat stellt CLOS einen Mechanismus zur Verfügung, der es ermöglicht, Klassen zu redefinieren. Dabei wird die Klasse unter Beibehaltung ihrer Identität so verändert, daß sie der neuen Definition entspricht. Alle abhängigen Objekte, also die direkten Instanzen,

[24] Alle Leser, die nicht sofort die Lösung parat hatten, seien getröstet: als ich dieses Beispiel Gregor Kiczales zeigte, ging es ihm nicht anders. ;-)

die direkten und indirekten Subklassen sowie ihre Instanzen, werden automatisch angepaßt. Der Zeitpunkt der Anpassung ist nicht genau spezifiziert, spätestens jedoch wenn ein abhängiges Objekt auf irgendeine Weise aktiviert wird, also bei Aufruf einer generischen Operation oder bei einem Slotzugriff. Für unser Beispiel heißt das, daß `point1` automatisch einen weiteren Slot `z` mit dem Wert `3` erhält:

```
(point-z point1)
; ==> 3
```

Der Benutzer hat die Möglichkeit, die vom System automatisch durchgeführte Anpassung der Instanzen zu beeinflussen bzw. zu ergänzen, indem er eine spezielle Methode der generischen Funktion `update-instance-for-redefined-class` für die redefinierte Klasse (hier `point`) hinzufügt. In der Regel wird dies eine *After-Methode* sein.

```
(defmethod update-instance-for-redefined-class :after
           ((p point) added deleted plist &rest initargs)
   (format t "~S has been updated." p))

(point-z point2)
; #<BLUE-POINT 0 0> has been updated.
; ==> 3
```

Zu beachten ist, daß die definierte Anpassungsmethode auch für Instanzen der Subklassen von `point`, hier `blue-point`, ausgeführt wird. Die systemdefinierte *Primary-Methode* von `update-instance-for-redefined-class` ruft die generische Funktion `shared-initialize` auf, um neu hinzugekommene Slots gemäß ihren Default-Werten zu initialisieren. Slots, die das Redefinieren überdauern, behalten ihre aktuellen Werte.

Der Mechanismus des Redefinierens von Klassen sorgt für ein gewisses Mindestmaß an konsistenter Anpassung der betroffenen Instanzen. Meistens ist eine anwendungsspezifische Anpassung erforderlich. Es ist offensichtlich, daß dieses Konzept den Einsatz traditioneller Kompilationstechniken erschwert, ja sogar verhindert. So nützlich es im Programmentwicklungsprozeß erscheint, so problematisch ist es im Hinblick auf die Effizienz und Robustheit schlüsselfertiger Applikationen. Es ist daher zu fragen, ob Redefinieren von Klassen wie in CLOS wirklich zur Standardfunktionalität eines Objektsystems zählen soll.

Reklassifizieren von Instanzen

In den meisten objektorientierten Programmiersprachen behält jedes Objekt seine Klassenzugehörigkeit während seiner gesamten Lebensdauer. Erfordert eine Anwendung die dynamische Zuordnung von Objekten zu bestimmten Anwendungsklassen zur Laufzeit des Programms, so können solche Klassen nicht auf Klassen der Programmiersprache abgebildet werden bzw. kann die (Anwendungs-)Klassenzugehörigkeit solcher Objekte nicht über ihre Instanz-von-Beziehung der Programmiersprache ausgedrückt werden. In CLOS gelten solche Beschränkungen nicht. Erleichtert wird dies dadurch, daß es keine strenge Typisierung von Variablenbindungen in CommonLisp gibt. CLOS bietet die Möglichkeit, die Klassenzugehörigkeit eines Objekts durch expliziten Aufruf der generischen Funktion `change-class` zu ändern, wie die folgende Beispielsfortsetzung zeigt:

```
(class-name (class-of point1))
; ==> POINT

(color-of point1)
; ==> DEFAULT-COLOR

(change-class point1 'blue-point)

(color-of point1)
; ==> BLUE
```

Das Objekt `point1` war bisher direkte Instanz der Klasse `point`, seine Farbe war `default-color`. Nun wurde sie unter Beibehaltung ihrer Identität zur direkten Instanz von `blue-point` und ihre Farbe wechselte entsprechend auf `blue`. Auch hier stellt CLOS eine generische Funktion `update-instance-for-different-class` zur Verfügung, die vom Benutzer spezialisiert werden kann, um den Klassenwechsel auf konsistente Weise abschließen zu können. Die systemdefinierte Primary-Methode sorgt dafür, daß die Struktur des Objekts der neuen Klasse entspricht. Slots, die den Klassenwechsel überdauern, behalten ihre aktuellen Werte, alle anderen neuen Slots werden gemäß ihren Ersatzwerten initialisiert.

```
(defmethod update-instance-for-different-class :after
           ((copy-with-previous-values point) changed-point &rest args)
   (format t "Changed class from ~S to ~S."
           (class-name (class-of copy-with-previous-values))
           (class-name (class-of changed-point))))

(change-class point2 'point)
; ==> Changed class from BLUE-POINT to POINT.
;     #<POINT 0 0>

(color-of point2)
; ==> DEFAULT-COLOR
```

Das Beispiel zeigt, daß Objekte nicht nur spezieller werden können (`point1` weiter oben), sondern auch allgemeiner: aus einem blauen Punkt ist ein einfacher Punkt geworden, auch seine Farbe hat sich geändert.

CLOS-Metaobjektprotokoll

Nachdem nun die wesentlichen Elemente der *Objektebene* von CLOS vorgestellt wurden, kann die *Metaobjektebene* betreten werden. [Kiczales *et al.*, 1991, S. 13] vergleichen ihre Systemarchitektur (*metalevel architecture*) mit einem Theater: die Objektebene ist die für die Zuschauer sichtbare Bühne, während die Metaobjektebene all das beinhaltet, was hinter den Vorhängen und Kulissen passiert, um das auf der Bühne Sichtbare und Hörbare stattfinden zu lassen. Die Objektebene stellt objektorientierte Sprachmittel für den Anwendungsprogrammierer zur Verfügung, die Metaobjektebene öffnet dem Systemprogrammierer eine abstrakte Sicht auf die Realisierung dieser Sprachmittel. Ein in der Informatik üblicher Trick dabei ist, die Realisierung selbst objektorientiert zu entwerfen, d. h.,

mit Mitteln der Objektsprache zu beschreiben. Da man auf diese Weise einen zirkulären Schluß (engl. *meta-circularity*) konstruiert hat, muß man besondere Vorkehrungen treffen, wie man das System an den eigenen Haaren aus dem Sumpf zieht (engl. *bootstrapping*). In diesem Zusammenhang spricht man auch von *reflektiven* oder *metazirkulären Systemen*, die in Lisp eine lange Tradition haben [Smith, 1982], [Smith, 1984], [Yonezawa und Smith, 1992] und bzgl. anderer Aspekte aus der Programm-Daten-Äquivalenz schon sehr früh abgeleitet wurden.

Prinzipiell können die Objektsprache und die Metaobjektsprache völlig unterschiedlich sein. Hier werden beide Sprachen in Lisp eingebettet, sie sind syntaktisch und semantisch uniform. Um so wichtiger ist es, sie als Programmierer dennoch auseinander halten zu können und dies auch zu tun, was in CommonLisp nur schlecht gelingt.

Elemente der Objektebene sind Definitionen von Klassen, generischen Funktionen und Methoden sowie Klasseninstanzen (Objekte) und Funktionsanwendungen. Auf der Metaebene wird nun eine Klassendefinition durch ein *Klassenobjekt*, eine generische Funktionsdefinition durch ein generisches Funktionsobjekt, eine Methode durch ein Methodenobjekt usw. repräsentiert. Diese werden *Metaobjekte* genannt. Sie bestimmen die Ausführung objektorientierter Programme, also z. B. wie auf Slots einer Instanz zugegriffen wird.

Das CLOS MOP ist funktional in drei Schichten eingeteilt. Die obere Schicht behandelt die syntaktischen Sprachkonstrukte der Objektebene: `defclass`, `defgeneric` und `defmethod`. Die mittlere Schicht sorgt für die Assoziierung von Namen mit Metaobjekten: `ensure-class` und `ensure-generic-function`[25]. Die dritte Schicht macht den eigentlichen Kern des MOP aus. Hier stellen systemdefinierte Metaobjekte ihre Funktionalität direkt zur Verfügung.

Bei der Betrachtung der MOP-Funktionalität werden statische und dynamische Aspekte unterschieden [Paepcke, 1993, S. 66]. Statische Aspekte umfassen die definierte Klassenhierarchie mit den Strukturanteilen von Metaobjektklassen. Dynamische Aspekte machen das Verhalten von Metaobjekten aus und werden in Form von Protokollen beschrieben. Diese spezifizieren, welche generischen Funktionen mit welchen Methoden für bestimmte Klassen vordefiniert sind, wie ihre dynamischen Aufrufstrukturen sind und welche Untergruppen sie bilden. Die kognitive Komplexität eines MOP ergibt sich aus den zirkulären Abhängigkeiten und der Schwierigkeit, unabhängige Untergruppen zu bilden bzw. geeignete Schnittstellen zwischen den Untergruppen zu spezifizieren.

Die vereinfachte Klassenhierarchie des CLOS MOP kann durch Einrückung wie folgt dargestellt werden:

```
T
  standard-object
    metaobject
      class
        built-in-class
        forward-referenced-class
        standard-class
        structure-class
```

[25] Wie schon einige andere Operationen, die der Metaobjektebene zuzuordnen wären (z. B. `class-of`, `class-name` etc.), werden diese in CLOS zur Objektebene gezählt.

```
  funcallable-standard-class
slot-definition
  standard-slot-definition
    standard-direct-slot-definition
    standard-effective-slot-definition
generic-function
  standard-generic-function
method
  standard-method
    standard-accessor-method
      standard-reader-method
      standard-writer-method
method-combination
```

Es werden also im wesentlichen fünf Arten von Metaobjekten unterschieden: Klassen, Slotdefinitionen, generische Funktionen, Methoden und Methodenkombinationen. In ObjvLisp waren es nur Klassen. Alle Metaobjektklassen außer `built-in-class` können vom Benutzer spezialisiert werden. Nun komme ich zum dynamischen Teil des Metaobjektprotokolls, das sich in folgende Subprotokolle einteilen läßt.

Das *Introspektionsprotokoll* erlaubt den lesenden Zugriff auf die jeweiligen Metaobjekte und stellt entsprechende generische Funktionen zur Verfügung. Ein schreibender Zugriff wurde nicht spezifiziert. Ob die mit Metaobjekten assoziierten Informationen als Slots repräsentiert werden oder nicht, ist nicht spezifiziert. Insgesamt muß wegen der Möglichkeit, Klassen zu redefinieren und Methoden zu entfernen, relativ viel der im Quellcode spezifizierten Information in den Metaobjekten gespeichert werden. Beispielsweise bräuchte man in einem Klassenobjekt, nachdem es initialisiert wurde, nur noch die Ergebnisse der dabei durchgeführten Vererbungsberechnungen zu speichern. Die direkt für eine Klasse spezifizierten Optionen wie die direkten Superklassen, die direkten Slotannotationen usw. könnte man vergessen. Da sich aber eine Superklasse dynamisch ändern könnte, indem sie redefiniert wird, muß ein Klassenobjekt in der Lage sein, sich erneut zu initialisieren, als ob ihr Quellcode erneut ausgewertet würde, ohne daß dies aber tatsächlich geschieht.

Das *Initialisierungsprotokoll* von Metaobjekten hat also als erstes dafür zu sorgen, daß die vom Introspektionsprotokoll benötigten Daten gespeichert werden. Anschließend muß die Funktionsfähigkeit anderer Subprotokolle hergestellt werden. Bei Klassen bedeutet dies, daß alle vererbungsrelevanten Berechnungen durchzuführen sind. Da Superklassen nicht unbedingt vor ihren Subklassen definiert sein müssen, wird dies hinausgezögert und im *Vererbungsprotokoll* zusammengefaßt, das erst bei Bedarf durch die generische Funktion `finalize-inheritance` angestoßen wird.

Im ersten Schritt des Vererbungsprotokolls wird die Vererbungshierarchie für die betroffene Klasse linearisiert (`compute-class-precedence-list`). Im zweiten Schritt werden alle Slots für die Instanzen dieser Klasse bestimmt (`compute-slots`). Dabei müssen auch die geerbten Slots und ihre Slotoptionen berücksichtigt werden. Im letzten Schritt wird die Klassenoption `default-initargs` behandelt: die generische Funktion `compute-default-initargs` führt ihre Vererbung durch.

Nach Abschluß des Vererbungsprotokolls müssen alle abhängigen Objekte, z.B. Subklassen, benachrichtigt werden (`update-dependent`). Gegebenenfalls müssen Instanzen, die

vor dem letzten Abschluß des Vererbungsprotokolls erzeugt wurden, als obsolet markiert werden (`make-instances-obsolete`), da ihre Klasse wegen des Redefinierens ggf. erneut initialisiert wurde und z. B. die Anzahl der Slots in den Instanzen nicht mehr mit der Slotanzahl in der Klasse übereinstimmt.

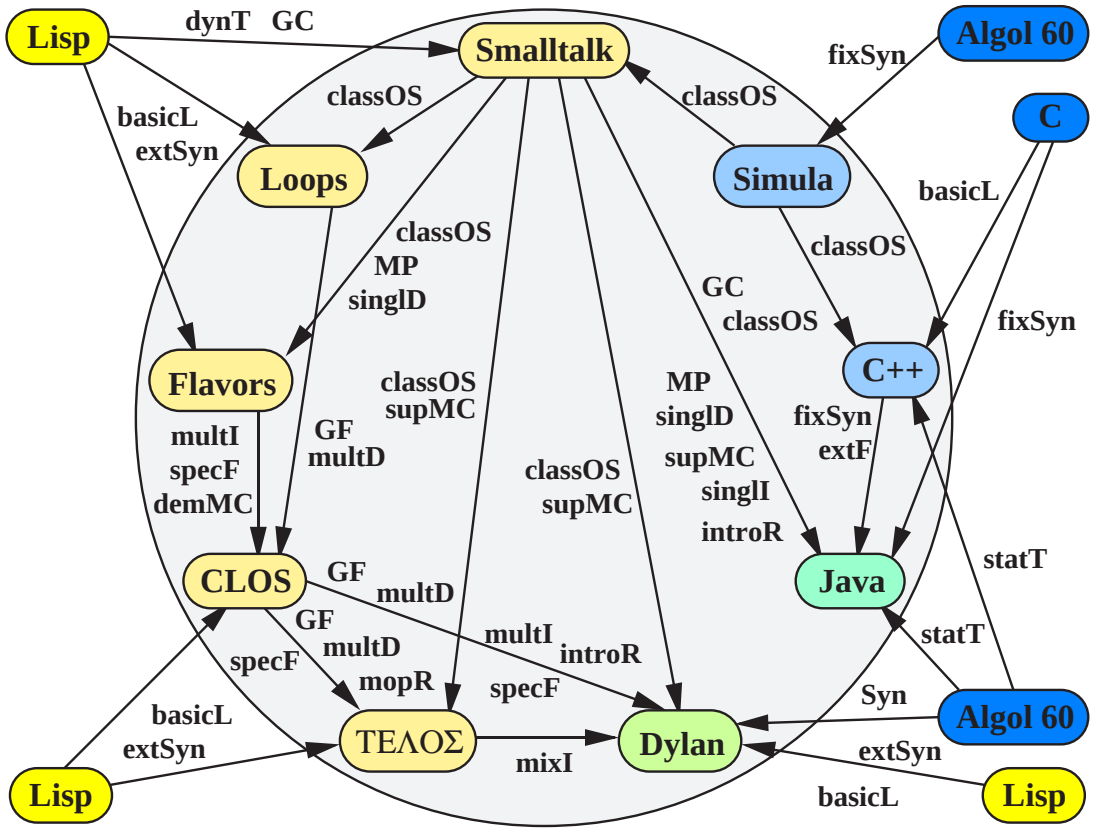
Das *Slotzugriffsprotokoll* benutzt die Ergebnisse der Berechnungen von `compute-slots` aus der Klasseninitialisierung und Vererbung. Dort wurde ermittelt, wie und wo der Slot alloziiert wird, z. B. in der Instanz oder in der Klasse und an welcher Position. Die allgemeinen Slotzugriffsoperationen der Objektebene `slot-value` und `(setf slot-value)` rufen zum Anwendungszeitpunkt die generischen Funktionen der Metaebene `slot-value-using-class` bzw. `(setf slot-value-using-class)` auf. Diese finden entsprechende Informationen in den Slotdefinitionsobjekten der Klasse und lesen bzw. schreiben den Slotwert einer Instanz. Da diese Vorgehensweise an so einer zeitkritischen Stelle zu aufwendig ist, müssen gute Implementierungen Optimierungen vornehmen. Eine Standardtechnik ist dabei, den Protokollpfad für eine Klasse (und Slot) nur beim ersten mal voll zu durchlaufen und beim nächsten mal möglichst viel abzukürzen (*caching*). Im Zusammenhang mit dem Redefinieren von Klassen muß aber ggf. wieder deoptimiert werden, was eine aufwendige Abhängigkeitsverwaltung erfordert.

Der *generische Dispatch* und die *Methodenauswahl* unterliegen ebenso einem spezifizierten Protokoll. Jeder generischen Funktion wird in Abhängigkeit von den mittels `add-method` hinzugefügten Methoden eine sogenannte Dispatch-Funktion zugeordnet. Diese wird ausgeführt, wenn die generische Funktion ausgeführt werden soll. Der Dispatch-Funktion werden dabei alle aktuellen Argumente des Aufrufs der generischen Funktion übergeben. In Abhängigkeit von den aktuellen Argumenten muß die Dispatch-Funktion die anwendbaren Methoden auswählen, ausführen und das Ergebnis zurückliefern. Die Dispatch-Funktion wird mittels `compute-discriminating-function` berechnet. Da Methoden einer generischen Funktion mit `remove-method` auch entfernt werden können, muß die Dispatch-Funktion ggf. mehrmals berechnet werden. Die Methodenauswahl erfolgt über `compute-applicable-methods` und `compute-applicable-methods-using-classes`. Die anwendbaren Methoden werden mit `compute-effective-method` zu einer ausführbaren Methode so zusammengefaßt, daß die jeweilige Methodenkombinationsart beachtet wird. Auch hier muß wie beim Slotzugriff optimiert werden [Kiczales und Rodriguez, 1990].

Alle genannten generischen Funktionen des CLOS MOP können vom Metaprogrammierer spezialisiert werden. Dabei sind gewisse Richtlinien und Einschränkungen zu beachten. Diese konnten leider nur in Form verbaler Beschreibungen [Kiczales *et al.*, 1991, S. 143] spezifiziert werden, ein Nichtbeachten führt zu unspezifiziertem Verhalten. Besser wäre es, schon durch die Konstruktion für gutartige Eigenschaften zu sorgen. Dieser Weg wurde in ΤΕΛΟΣ gewählt.

3.5 Zusammenfassung

Vergleicht man die vorgestellten objektorientierten Programmiersprachen zusammenfassend miteinander, so ergibt sich das Verwandschaftsbild in Abbildung 3.3. Dabei habe ich im Vorgriff auf die folgenden Kapitel breits ΤΕΛΟΣ und DYLAN in die Darstellung aufgenommen. Letztere zeigen eine gewisse Konvergenz der bereitgestellten Konzepte.



basicL	Basissprache
statT	statische Typisierung
dynT	dynamische Typisierung
GC	Garbage Collection
extSyn	erweiterbare Syntax
fixSyn	nicht-erweiterbare Syntax
classOS	klassenbasiertes Objektsystem
singlI	einfache Vererbung
multI	multiple Vererbung
mixI	Mixin-Vererbung
extF	erweiterbare Felder
specF	spezialisierbare Felder
MP	Message Passing
GF	generische Funktionen
singlD	einfacher Dispatch
multD	mehrfacher Dispatch
supMC	einfache Methodenkombination
demMC	Demon-Methodenkombination
mopR	Metaobjektprotokoll
introR	Introspektionsprotokoll

Abbildung 3.3: Konzeptuelle Verwandschaft objektorientierter Programmiersprachen.

Alle vorgestellten Sprachen sind dem Beispiel SIMULA folgend klassenbasiert, im Unterschied zu Actors und Self, auf die ich nicht näher eingegangen bin. C++ kommt als einzige Sprache ohne automatische Speicherbereinigung aus. Und auch sonst fällt auf, daß C++ die wenigsten Konzepte aus anderen Sprachen übernommen hat. Auffällig ist, daß JAVA zwar syntaktisch C++ ähnlich ist, seine objektorientierten Konzepte jedoch überwiegend von Smalltalk stammen.

Die Tabelle 3.1 zeigt eine qualitative Bewertung der Sprachen anhand der Designkriterien aus Kapitel 2.

Kriterium	Smalltalk	C++	Java	CLOS
Abstraktion	+	– –	–	++
Klassifikation	+	+	+	++
Spezialisierung	+	– –	–	++
Komposition	+	– –	–	+
Modularisierung	–	–	–	–
Orthogonalität	–	– –	–	–
Einfachheit	–	– –	+	– –
Reflektion	+	– –	+	++
Erweiterbarkeit	–	– –	–	++
Robustheit	–	–	+	–
Effizienz	–	++	–	–
Verursacherprinzip	–	+	+	–

Tabelle 3.1: Vergleich der Sprachen anhand der Designkriterien.

Gemessen an meinen Designkriterien bietet CLOS im Vergleich zu Smalltalk, C++ und JAVA die besten Abstraktions-, Klassifikations- und Spezialisierungsmittel. Hier sind die spezialisierbaren Felder, generische Funktionen mit mehrfachem Dispatch und die Methodenkombination zu nennen. Sowohl die syntaktische als auch die funktionale Erweiterbarkeit wird in CLOS mit Makros und mit dem Metaobjektprotokoll am besten unterstützt. Smalltalk und JAVA bieten zwar Mittel zur Introspektion, sind aber syntaktisch nicht erweiterbar und bieten kein Metaobjektprotokoll. C++ fällt hier noch weiter zurück. Die Schwächen von CLOS liegen bei den Kriterien Modularisierung, Orthogonalität und Einfachheit, worunter fast zwangsläufig die Robustheit und Effizienz der Sprache leidet. Auch Smalltalk ist hierbei schwächer als C++ und JAVA. Zwar gilt C++ als unschlagbar in der Effizienz, seine Robustheit kann jedoch ohne automatische Speicherverwaltung und mit den erlaubten Pointermanipulationen nicht zufriedenstellen. Ebenso unzulänglich wird das Generalisieren und Spezialisieren der Struktur und des Verhaltens von Objekten unterstützt, weil weder spezialisierbare Felder, noch mehrfacher generischer Dispatch noch Methodenkombination bereitgestellt werden. Smalltalk und JAVA bieten zwar einfache Methodenkombination (das Konzept `super()`), aber keinen mehrfachen Dispatch. Insbesondere die Dispatch-Regeln von JAVA sind kontraintuitiv und erschweren das Generalisieren und Spezialisieren von Verhalten. Dies liegt zum Teil auch an der statischen Typisierung von Variablen in JAVA, was in Smalltalk nicht gegeben ist. Auf der anderen Seite erhöht das *Compile-Time-Type-Checking* in JAVA natürlich die Robustheit von

Programmen.

Es stellt sich nun die Frage, ob und wie die besseren Konzepte von CLOS so realisiert werden können, daß es möglich wird seine Schwächen zu überwunden. In den folgenden Kapiteln werde ich die Konzepte so überarbeiten, daß darauf eine positive Antwort gegeben werden kann.

Abschließend zeigt die Tabelle 3.2 die bereitgestellten Sprachkonzepte im Überblick.

Kürzel	Sprachkonzept	Smalltalk	C++	Java	CLOS
Vererbung					
singlI	einfache Vererbung	ja	nein	ja	nein
multI	multiple Vererbung	nein	ja	nein	ja
mixI	Mixin-Vererbung	nein	nein	nein	nein
Struktur					
instF	instanzspezifische Felder	ja	ja	ja	ja
classF	klassenspezifische Felder	ja	ja	ja	ja
extF	erweiterbare Felder	ja	ja	ja	ja
specF	spezialisierbare Felder	nein	nein	nein	ja
Verhalten					
MP	Message Passing	ja	ja	ja	ja
GF	generische Funktionen	nein	nein	nein	ja
singlD	einfacher Dispatch	ja	ja	ja	ja
multD	mehrfacher Dispatch	nein	nein	nein	ja
MC	Methodenkombination	super	nein	super	super/demon
classD	klassenspezifischer Dispatch	ja	ja	ja	ja
instD	instanzspezifischer Dispatch	nein	nein	nein	ja
Dynamik					
dynC	Klassenwechsel	ja	nein	nein	ja
redC	redefinierbare Klassen	ja	nein	nein	ja
Reflektion					
funR	funktionale Objekte	ja	nein	nein	ja
introR	Introspektion	ja	nein	ja	ja
mopR	Metaobjektprotokoll	nein	nein	nein	ja
Orthogonale Aspekte					
Syn	Syntax	speziell	C	C++	Lisp
extSyn	erweiterbare Syntax	nein	nein	nein	ja
GC	automatische Speicherbereinigung	ja	nein	ja	ja
Mod	Modularisierung	Klasse	Datei Klasse	Klasse Package	Datei Package
Hid	Sichtbarkeit	Klasse	Datei Klasse	Klasse Package	Package
bindMod	Modularisierung des Bindungsraums	Block	Block	Block	Block
nameMod	Modularisierung des Namensraums	-	Klasse	Package	Package

Tabelle 3.2: Vergleich der bereitgestellten Konzepte objektorientierter Programmiersprachen.

Kapitel 4

Objektorientierte Sprachkonzepte

Design a language; don't just hack one up!

Frederick P. Brooks, Jr. 1993. Keynote address: Language Design as Design, The Second History of Programming Languages Conference (HOPL-II), [Brooks, 1996].

Nach der Diskussion des Stands der Technik möchte ich nun meine Sicht objektorientierter Sprachkonzepte formulieren und den Entscheidungsspielraum beim Entwurf entsprechender Sprachkonstrukte aufspannen. Im nächsten Kapitel werden dann die Implementierungstechniken diskutiert, bevor in den darauffolgenden Kapiteln drei konkrete Objektsysteme schrittweise entwickelt und die jeweils getroffenen Designentscheidungen begründet werden. Als Notation für die Sprachkonzepte wird in diesem Kapitel eine Java-ähnliche Syntax[1] verwendet, um ihre über Lisp hinausgehende generelle Gültigkeit zu unterstreichen. Die von mir entwickelten Konzepte können für eine traditionell kompilativ implementierte objektorientierte Sprache ebenso umgesetzt werden wie für Lisp. Selbst die Konzepte der Metaobjektprogrammierung können durch eine Ergänzung des Laufzeitsystems traditioneller Sprachen um einen kleinen Metaobjektkern mit Objekten und Operationen einfach integriert werden. Der Compiler kann dann den entsprechenden Code aus Anwendungen der Metaobjektprogrammierung erzeugen, der sich auf den Kern des Laufzeitsystems stützt. Die konkrete Ausprägung der Metaobjektprotokolle, wie sie hier vorgestellt werden, setzt lediglich *funktionale Objekte erster Ordnung* voraus, wie sie in allen funktionalen Sprachen vorhanden sind. Aber selbst darauf könnte im Prinzip verzichtet werden.

Das Kapitel beginnt mit einer kurzen Diskussion orthogonaler Sprachaspekte. Dabei soll zum einen klargemacht werden, warum einige, möglicherweise erwartete, Sprachkonzepte nicht als objektorientiert im Sinne dieser Arbeit eingeschätzt werden. Zum anderen sollen einige Voraussetzungen bzw. Ergänzungen von Objektsystemen genannt werden.

Die Darstellung meiner Sicht objektorientierter Sprachkonzepte intendiert bereits ihre Strukturierung und Modularisierung. Objekt- und Metaobjektebenen werden strikt getrennt.

[1] Die genaue Syntaxspezifikation ist in [Shalit, 1996] zu finden.

4.1 Orthogonale Sprachaspekte

Um Mißverständnissen vorzubeugen sei ausdrücklich erklärt, daß folgende orthogonale Sprachkonzepte ihre eigenständige Bedeutung für eine Programmiersprache haben. Ihre Präsenz in einer objektorientierten Sprache erfordert eine sorgfältige Abstimmung der Konzepte aufeinander. Ihre Abgrenzung vom Objektsystem läßt das Objektsystem selbst schlanker werden, weil einige Sprachaspekte an einer passenderen Stelle schon behandelt werden.

4.1.1 Verteilte Systeme und Kommunikation

Eine moderne universelle Programmiersprache sollte geeignete Mittel für verteilte Systeme und Kommunikation bereitstellen. Dies kann von einfachen Konzepten wie Coroutinen, über Prozesse und Semaphore, Monitore etc. bis hin zu Message Passing und Netzprotokollen reichen. In einer objektorientierten Programmiersprache können solche Konzepte auch objektorientiert zur Verfügung gestellt werden [Christaller, 1987], z. B. durch entsprechende Systemklassen. Daraus muß aber nicht folgen, daß alle Objekte die Komplexität und die Kosten verteilter Systeme aufgebürdet bekommen sollen. Insofern lassen sich Aspekte der Nebenläufigkeit und Synchronisation so eingrenzen, daß die Wechselwirkungen mit den objektorientierten Sprachmitteln, wie sie hier gesehen werden, eher gering sind. In Kapitel 8 werden die Wechselbeziehungen am Beispiel EuLisp erörtert. Genauer geht dieser Fragestellung Queinnec in [1995] nach. Auch Java [Gosling *et al.*, 1996] kann als Beispiel für eine arbeitsteilige Konzeption von Sprachmitteln zur Objektorientierung (Klassifizierung) und solcher zur Nebenläufigkeit und Synchronisation angesehen werden.

Die Metapher des Nachrichtenversendens bzw. das Kommunikationsmodell wird nicht als Ausgangsparadigma für das Objektsystem verwendet. Aspekte verteilter Systeme und ihrer Kommunikation betrachte ich als orthogonal zur Objektorientierung im Sinne der Klassifizierung, Spezialisierung und Generalisierung von Objekten. Für die meisten Objekte spielen Verteiltheit, Kommunikation und Synchronisation keine Rolle. Daher soll der Spezialfall nicht zur Regel erklärt werden. Natürlich können spezielle Anwendungen auch durchgehend nach dem Kommunikationsmodell organisiert sein.

4.1.2 Dynamische Speicherverwaltung

Dynamische Speicherverwaltung stellte für viele Anwendungen lange Zeit ein großes Problem dar. Für einige Sprachen wie Lisp stellten die entsprechenden Laufzeitsysteme zwar schon früh eine automatische Speicherverwaltung (engl. *garbage collection*) bereit. Die ursprünglichen einfachen *Mark&Sweep-Algorithmen* reichten in ihrer Performanz aber für zeitkritische, insbesondere echtzeitfähige System, zunächst nicht aus. Andererseits ist die manuelle Verwaltung des dynamischen Speichers in größeren Anwendungen sehr fehleranfällig und erschwert ihre Wartung: ca. 60 Prozent aller Fehler in C-Programmen sind Pointer-Fehler, wovon ein Großteil mit manueller Verwaltung des dynamischen Speichers zusammenhängt. Objektorientierte Sprachmittel verschärfen das Problem, weil auf der einen Seite der Speicherbedarf wegen zur Laufzeit benötigter Typinformation und vieler zur Laufzeit erzeugter Instanzen ansteigt. Auf der anderen Seite steigt in objektorientier-

ten Sprachen der Verknüpfungsgrad von unterschiedlichen Objekten, so daß eine manuelle Speicherverwaltung kaum noch fehlerfrei programmiert werden kann.

Dank der Fortschritte im Hardwarebereich und der entwickelten Speicherbereinigungsverfahren (engl. *Garbage-Collection* oder abgekürzt GC) [Ungar, 1988], [Moon, 1984], [Boehm und Weiser, 1988], [Kriegel, 1992] besteht aber immer weniger Grund auf automatische Speicherverwaltung zu verzichten. Smalltalk, Lisp, Eiffel, Java und Dylan machen es mit Erfolg vor. Und selbst Stroustrup schließt die Integration eines Garbage-Collectors in C++ nicht mehr aus [Stroustrup, 1994, S. 220]. Beispielsweise ist der Garbage-Collector für EuLisp aus dem Projekt APPLY in Wirklichkeit sprachunabhänig und kann unverändert auch für C++ benutzt werden [Kriegel, 1992].

4.1.3 Persistenz

Von *persistenten Objekten* spricht man, wenn bestimmte Objekte eine längere Lebensdauer haben sollen, als die durch die Programmausführung und Terminierung vorgegebene Zeitspanne. Wenn das Programm erneut gestartet wird, soll es nicht erneut erzeugt werden müssen, sondern es soll wieder so vorgefunden werden, wie es bei der Programmterminierung war. Hier ist der Bezug zu Datenbanken offensichtlich. Nun sind traditionelle Datenbanken in der Regel relational, also nicht objektorientiert. Die Kopplung einer objektorientierten Programmiersprache mit einer Datenbank ist technisch nicht ganz trivial. Deshalb wurden ad hoc einfachere Lösungen für persistente Objekte entwickelt, indem diese auf Dateien geschrieben und später eingelesen werden. Doch inzwischen bieten auch viele Datenbanksysteme objektorientierte Schnittstellen an, so daß nicht nur Persistenz von Objekten erreicht werden kann, sondern auch flexible Abfragesprachen zur Verfügung stehen. Auf der Objektsystemseite kann Persistenz am besten mit Hilfe des Metaobjektprotokolls realisiert werden [Paepcke, 1988], [Paepcke, 1990]. In meiner Arbeit wird Persistenz zwar nicht weiter thematisiert, es ist aber sichergestellt, daß das Metaobjektprotokoll von Τελος seine Bereitstellung unterstützt [Broadbery und Burdorf, 1993].

4.1.4 Modularisierung und Sichtbarkeit

Klassische Modulkonzepte unterstützen die Aufteilung größerer Programme in kleine bis mittlere *Bausteine* (Blockbildung) mit strikten *Import- und Export-Schnittstellen*. Typischerweise kann auch zwischen *Spezifikation* und *Implementierung* unterschieden werden. *Sichtbarkeit* und *Gültigkeit* von Bezeichnern bzw. Variablen- und Konstantenbindungen beziehen sich auf Module. Modulgrenzen dürfen nur durch die Import-/Export-Tore überschritten werden. *Kapselung* kann man als dual zur Sichtbarkeit auffassen: was nicht sichtbar ist, ist gekapselt. Weiterhin bilden Module eigenständig *übersetzbare Einheiten* und können zu sogenannten Bibliotheken (*libraries*) zusammengefaßt werden. Als Beispiele für Sprachen, die modulares Programmieren unterstützen, sind Modula-2 und Ada zu nennen.

Der Streit darüber, ob Klassen gleichzeitig auch die Rolle von Modulen übernehmen können und sollen, ist noch nicht entschieden. In Smalltalk und Eiffel müssen Klassen weitgehend auch Aspekte von Modulen (wie in Modula-2) abdecken. Allerdings gelingt dies nicht vollständig. So gibt es in Smalltalk auch noch sogenannte *Pools*, um klassenübergreifend sichtbare Variablen zu ermöglichen. In Eiffel gibt es dafür sogenannte *Units*. C++

und Java nehmen eine pragmatische Zwitterstellung ein: Klassen müssen regeln, was sie nach außen sichtbar machen, aber sie importieren nichts. Da helfen in C++ die aus C bekannten Konventionen mit den sogenannten *Header-Dateien* mit *Include-Anweisungen*, während Java zusätzlich das Konzept der *Packages* einführt, um Sichtbarkeit für Gruppen von Klassen zu regeln, und zwar zusätzlich zu dem, was in der Klassendefinition schon geregelt ist. Import-Anweisungen sind aber auch nicht auf Packages bezogen, sondern auf die Quellcode-Datei, also ähnlich wie in C++.

EuLisp und dylan teilen die Auffassung von [Szyperski, 1992], wonach man beides braucht: Klassen und Module als orthogonale Konzepte. Auch [Booch, 1994] vertritt diese Sicht, obwohl er Kapselungsaspekte (also Sichtbarkeit) dann doch den Klassen zuschreibt. Konsequenter ist es, das Objektsystem gänzlich von Sichtbarkeits- und Modularisierungsaspekten zu entlasten, da dies das Modulkonzept besser regeln kann. Diese Sicht lege ich dieser Arbeit zugrunde. Dies muß nicht unbedingt dogmatisch gesehen werden. Es ist aber eine so grundsätzliche Entscheidung, daß sie am besten von vornherein getroffen werden sollte, weil man sonst zu viele Fallunterscheidungen bei spezielleren Fragen machen müßte.

Ob die eine oder die andere Entscheidung getroffen wird, hängt wohl auch davon ab, ob es in einer Sprache Multimethoden gibt oder nicht. Alle genannten Sprachen ohne Multimethoden ordnen Sichtbarkeitsaspekte den Klassen zu. Methoden befinden sich dort lexikalisch (und textuell) innerhalb einer Klassendefinition und sind daher auch genau einer Klasse zugeordnet, obwohl es weitere Parameter anderer Klassen geben kann. CLOS, EuLisp und dylan unterstützen Multimethoden. Da diese über mehrere Argumente diskriminieren können, kann es auch nicht immer eine plausible Zuordnung zu genau einer Klasse geben. Konsequenterweise enthalten Klassendefinitionen hier keine Methodendefinitionen, außer Lese- und Schreiboperationen. Es ist daher naheliegend, eine Hülle um ggf. mehrere Klassen und ihre Methoden zu legen, die aber selbst keinen neuen Objekttyp, und somit eine Klasse, konstituiert.

So kann man auch in Java typische Klassen identifizieren, die reinen Modulcharakter haben. Dazu gehören die sogenannten Hauptprogramm-Klassen (*main class*) mit der obligatorischen Haupt-Methode (*main method*) oder auch alle Klassen, die nur Klassenvariablen bzw. Klassenmethoden (*static*) besitzen und die nie instanziiert werden. Die Java-Klasse `java.lang.Math` ist zum Beispiel so eine Klasse, die übrigens auch nicht spezialisiert werden darf. Es gibt also gute Gründe zwischen Klassen- und Modulbildung zu unterscheiden, selbst wenn es in einer Sprache nur ein syntaktisches Konstrukt für beides gibt.

Auf jeden Fall muß eine objektorienterte Programmiersprache beides unterstützen: Modularisieren mit klaren Sichtbarkeitsregeln sowie Klassifizieren mit klarer Semantik aus Modellierungs- und Sprachimplementierungssicht. Hier setze ich ein zum Objektsystem orthogonales Modulsystem voraus. Das heißt nicht, daß das zu entwerfende Objektsystem ohne ein Modulsystem nicht funktionsfähig wäre. Vielmehr werden bestimmte Anforderungen aus dem Software-Engineering wie das Modularisierungskriterium aus Abschnitt 2.3.6 oder Informationskapselung (*information hiding* und *encapsulation*) an ein hoffentlich vorhandenes Modulsystem delegiert. Dadurch kann man sich beim Entwurf des Objektsystems auf das Essentielle konzentrieren. Die konkrete Annahme über das Modulkonzept ist, daß Variablen und Konstanten lexikalisch bezogen auf ein Modul gebunden werden. Bindungen werden von Modulen importiert und exportiert. Dabei können die jeweiligen Bezeichner umbenannt werden, um Namenskonflikte zu vermeiden.

Das Objektsystem selbst soll modular aufgebaut sein, insbesondere um eine klare Unterscheidung zwischen der Objekt- und der Metaobjektebene vorzunehmen. Aber auch innerhalb der Metaobjektebene ist es sinnvoll, die Unterprotokolle in entsprechende Module aufzuteilen.

4.1.5 Ausnahmebehandlung

Ausnahmebehandlung (*exception handling*) dient dazu, auf selten auftretende Ereignisse während der Programmausführung auf einem angemessen abstrakten Niveau reagieren zu können. Insbesondere erzeugen Laufzeitfehler Ausnahmen. Dabei unterbricht so ein Ereignis den normalen Kontrollfluß, die *Ausnahmesituation* (*exception*) wird gemeldet. Eine potentielle Reaktion darauf wird durch die Deklaration eines *Exception-Handlers* vorbereitet. Der Zusammenhang zwischen einer Ausnahmemeldung und ihrer Ausnahmebehandlung ist nicht durch die lexikalische Programmstruktur bestimmt, sondern ergibt sich aus dem dynamischen Programmablauf. In einer objektorientierten Sprache bietet es sich an, die verschiedenen Ausnahmesituationen zu klassifizieren. Einige Klassen werden vom System vordefiniert und können vom Benutzer spezialisiert werden. Ebenso kann die Reaktion objektorientiert erfolgen, indem man eine generische Operation aufruft, die über das Ausnahmeobjekt diskriminiert und so die spezifisch gewünschten Aktionen auslöst.

Für den Entwurf und die Implementierung des Objektsystems hat ein zu unterstützendes Konzept der Ausnahmebehandlung die Konsequenz, daß die Methodenausführung dynamisch unterbrechbar sein muß. Dies ist aber nichts neues im Vergleich zu funktionalen oder prozeduralen Aspekten einer Programmiersprache, die ebenso diese Forderung erfüllen müssen. Auf das konkrete Zusammenwirken von ΤΕΛΟΣ mit der EuLisp-Ausnahmebehandlung [Padget *et al.*, 1993] wird in Kapitel 8 eingegangen.

4.2 Klassifizieren der Struktur und des Verhaltens von Objekten

Wie im letzten Kapitel gezeigt wurde, gehen die meisten objektorientierten Programmiersprachen davon aus, daß die Identifizierung ähnlicher Objekte und ihre normative Beschreibung in Klassendefinitionen ein zentrales Sprachkonzept darstellen. Dabei werden Sprachkonstrukte benötigt, um die in der Regel *statische Struktur* und das meist vom aktuellen Objektzustand abhängige *dynamische Verhalten* zu beschreiben. Beide Aspekte fasse ich zu *Objekteigenschaften* zusammen. Die selteneren Situationen, daß die Struktur dynamisch ist bzw. das Verhalten statisch, sollen keineswegs ausgeschlossen werden. Aber sie dürfen die Regelfälle sowohl konzeptuell als auch von der zu erwartenden Performanz her nicht beeinträchtigen.

Man kann die Objekteigenschaften auch danach unterscheiden, ob sie *instanzspezifisch* oder *klassenspezifisch* sind. Würde man beispielsweise alle blauen, kreisförmigen Objekte zu einer Klasse zusammenfassen, so wäre ihre Farbe oder die Vorschrift zur Flächenberechnung klassenspezifisch, während die Mittelpunktkoordinate und der Radius instanzspezifisch wären. Meistens sind die Verhaltensbeschreibungen (Methoden) klassenspezifisch und die Zustandswerte (Felder, Attribute, Slots, Instanzvariablen etc.) instanzspezifisch.

Auch hier sollen die selteneren Fälle wie Klassenvariablen oder instanzspezifische Methoden (Eql-Methoden in CLOS oder *Singleton-Methoden* in DYLAN) konzetionell nicht ausgeschlossen sein, aber sie dürfen ebenfalls den Regelfall nicht erschweren.

Die Einteilung von Objekteigenschaften in Struktur und Verhalten spiegelt die Sicht auf das Innere eines Objekts wider. Die interne Zustandsrepräsentation wird im Laufe der Programmentwicklung häufiger geändert und sollte daher nach außen verborgen bleiben. Um die externe Schnittstelle eines Objekts möglichst unverändert lassen zu können, ist es ratsam, Lese- und Schreibzugriffe syntaktisch uniform mit Methodenaufrufen zu gestalten. Dann müssen im sie benutzenden Client-Code viel seltener Anpassungen vorgenommen werden, falls Felder hinzugefügt bzw. entfernt werden.

Da die Struktur und das Verhalten von Objekten klassifizierend beschrieben wird, bekommt das *Vererbungskonzept* eine zentrale Bedeutung. Alternative Vererbungskonzepte werden im nächsten Abschnitt ausführlich behandelt. An dieser Stelle sei aber schon festgelegt, daß

- es eine systemdefinierte Klasse aller Objekte mit dem Namen `Object` geben soll,
- jeder Klasse eine *Klassenpräzedenzliste* zugeordnet wird, die mit der Klasse selbst beginnt und mit `Object` endet; ihre Reihenfolge legt fest, welche Klassen *spezieller* und welche *genereller* sind;
- die Klassenpräzedenzliste die Vererbung von Struktur und Verhalten steuert, d. h. das *Initialisierungsprotokoll von Klassen* (statische Vererbung von Feldern) sowie den *Methoden-Dispatch* bei Anwendung generischer Operationen (dynamische Vererbung von Methoden).

Diese Festlegung ist speziell genug, um alle Objekte aus Systemsicht einheitlich zu behandeln, also z. B. systemdefinierte generische Operationen auf beliebige Objekte anzuwenden. Sie ist gleichzeitig allgemein genug, um ganz andere objektorientierte Konzepte als Spracherweiterung zu realisieren, also z. B. solche, die nicht eine Linearisierung der Vererbungshierarchie vornehmen und den Methoden-Dispatch auf andere Weise durchführen. Deren Klassenpräzedenzliste könnte dann nur aus der jeweiligen Klasse selbst und `Object` bestehen.

4.2.1 Struktur von Objekten

Die Struktur eines Objekts dient der Repräsentation seines Zustands. Sie impliziert eine entsprechende Art der Allozierung, Initialisierung sowie der Lese- und Schreiboperationen auf einem Objekt. Es wird deutlich, wie eng die Struktur und das Verhalten zusammenhängen.[2] In den meisten objektorientierten Programmiersprachen geht die Programmkonstruktion *bottom-up* vor sich, d. h. zuerst werden kleinere Bausteine definiert und dann werden sie für größere verwendet. In deklarativen Sprachen geht man häufiger umgekehrt, also *top-down* vom Ziel aus. Dies mag erklären, warum man im sequentiellen Text mit der Struktur anfängt und später zum Verhalten kommt, obwohl beides aufeinander bezogen

[2] In der Künstlichen Intelligenz spricht man in diesem Zusammenhang von Repräsentation und Inferenz.

ist und obwohl man sich als Programmierer schon vor der Wahl einer konkreten Struktur Gedanken über das gewünschte Verhalten macht.

Im einfachsten Fall beschreibt man die Objektstruktur durch Aufzählen seiner *Felder* in der Klassendefinition:

```
define class Point (Object)
  field x;
  field y;
  ...
end class Point;
```

Felder werden indirekt über Bezeichner, die an vom System generierte Namen gebunden werden, identifiziert. Im obigen Beispiel bedeutet dies, daß es im Modul `M1`, das die Klasse `Point` enthält, eine konstante Bindung gibt, die wie folgt definiert sein könnte:

```
define x = newName();     // ==> M1.Point.x  z.B.
```

Der Wert von `x` in Modul `M1` könnte anschließend das Symbol `M1.Point.x` sein[3]. In der Schnittstellenspezifikation von `M1` kann dann z. B. `x` exportiert, `y` aber verborgen werden. Ein exportierter Feldname kann nur dem Zweck dienen, in einem anderen Modul `M2` bei der Definition einer Subklasse von `Point` spezialisiert zu werden:

```
define class myPoint (Point)
  specialize field x ...;
  field y;
  ...
end class myPoint;
```

Bei Feld `x` würde es sich nun in beiden Klassen, `Point` und `myPoint`, um dasselbe Feld handeln. Das Feld `y` in der Klasse `myPoint` hat jedoch nichts mit dem von `Point` geerbten Feld, das im Module `M1` auch mit `y` bezeichnet wurde, zu tun. Somit nutze ich das Modulkonzept, um Kapselung der Objektstruktur zu unterstützen und Namenskonflikte, insbesondere bei Feldern, zu vermeiden. Dies ist übrigens eine ähnliche Lösung wie in Oz, wo generierte Namen als Werte lexikalischer Bindungen zur Identifizierung von Record-Feldern benutzt werden, um den Zustand von Objekten zu repräsentieren.

Will man nun Objekte um strukturbezogene Verhaltensaspekte anreichern, so führt man sogenannte *Feldannotationen* (Slotoptionen in CLOS) ein. Hierbei ist zu beachten, daß einige dieser Annotationen nicht unabhängig voneinander sind.

Um Objekte auf syntaktisch uniforme Weise anzusprechen (Operationsanwendung), können die Feldannotationen `reader:`, `writer:` und `accessor:` spezifiziert werden. Sie bewirken, daß mit der Klassendefinition automatisch entsprechende *Lese-* und *Schreiboperationen* erzeugt und an die spezifizierten Namen gebunden werden. Die Annotation

[3] Welcher Wert es ist, spielt eigentlich keine Rolle. Es muß nur sichergestellt sein, daß alle Ergebnisse der Operation `newName` verschieden sind, ähnlich wie die von `gensym` in Lisp. Diese konstante Bindung stellt lediglich eine Hilfskonstruktion dar. Ein optimierender Compiler könnte sie wieder entfernen und alle Referenzen auf `x` durch den konstanten Wert ersetzen.

`accessor:` bewirkt, daß eine Leseoperation mit dem angegebenen Namen und eine Schreiboperation mit dem gleichen Namen, ergänzt um das Präfix "set", generiert werden. Der auf "set" folgende Buchstabe ist dann in jedem Fall groß, also z. B. setPointX.

```
define class Point (Object)
  field x accessor: pointX;
  field y reader: pointY, writer: writePointY;
  ...
end class Point;

pointX(setPointX(99, make(Point)));  // ==> 99
```

Auf der Objektebene gibt es keine andere Möglichkeit auf Felder zuzugreifen als über spezifizierte Lese- und Schreiboperationen. Diese unterliegen den Sichtbarkeitsregeln des Modulkonzepts, können also auch importiert und exportiert werden. Insofern ist auch die Forderung nach Kapselung von Struktur voll erfüllt.

Um die Felder bei der Objekterzeugung mit *Ersatzwerten* zu initialisieren, wenn keine expliziten Initialisierungsargumente angegeben werden, können die Feldannotationen `initValue:`, `initFunction:` und `initExpression:` spezifiziert werden, aber für ein und dasselbe Feld höchstens eine von ihnen:

```
initialPosition := 0
define method computeVerticalPosition () 2; end method;

define class Point (Object)
  field x initValue: initialPosition + 1, reader: pointX;
  field y initFunction: computeVerticalPosition, reader: pointY;
  field z initExpression: initialPosition + 3, reader: pointZ;
...
end class Point;

initialPosition := 5;
pointX(make(Point));  // ==> 1 (0 + 1)
pointY(make(Point));  // ==> 2 (computeVerticalPosition())
pointZ(make(Point));  // ==> 8 (5 + 3)
```

Die Annotation `initValue:` bewirkt, daß der spezifizierte Ausdruck (hier: `initialPosition + 1`) genau einmal ausgewertet wird und zwar zur Klassendefinitionszeit. Bei Instanzerzeugung wird immer derselbe Wert genommen. Für `initExpression:` wird der spezifizierte Ausdruck (hier: `initialPosition + 3`) bei jeder Instanzerzeugung ausgewertet, so daß der jeweils aktuelle Wert von `initialPosition` zum Tragen kommt. Für `initFunction:` gilt das gleiche wie für `initExpression:`, nur daß die spezifizierte Funktion ausgeführt wird.

Will man dem allgemeinen Konstruktor `make` *Initialisierungsargumente* mitgeben, so können die Feldannotationen `initKeyword:` oder `requiredInitKeyword:` in der Klassendefinition spezifiziert werden. Initialisierungsargumente werden also über *Initialisierungs-Schlüsselwörter* identifiziert:

```
define class Point (Object)
  field x initKeyword: x:, initValue: 1, reader: PointX;
  field y requiredInitKeyword: y:, reader: PointY;
  ...
end class Point;

(make(Point);
// ==> ERROR: missing required argument y:

pointX(make(Point, y: 22));          // ==> 1
pointX(make(Point, x: 11, y: 22));   // ==> 11
pointY(make(Point, y: 22));          // ==> 22
```

Die Annotation `initKeyword:` ermöglicht, *optionale Initialisierungsargumente* anzugeben. Dagegen bewirkt `requiredInitKeyword:`, daß ein Initialisierungsargument gefordert wird, anderenfalls wird ein Fehler gemeldet. Ist ein *gefordertes Initialisierungsargument* spezifiziert, so wird ein *Ersatzwert* über `initValue:`, `initFunction:` oder `initExpression:` überflüssig.

Weiterhin kann es sinnvoll sein, den Typ möglicher Feldwerte einzuschränken, wofür es die Annotation `type:` gibt. Dabei stellt sich hier die Frage, ob es in der Programmiersprache nur Klassen als Repräsentationstypen gibt, oder auch z. B. Aufzählungs-, Bereichs- und Prädikationstypen sowie andere abgeleitete Typen unterstützt bzw. ermöglicht werden sollen, um eine höhere Programmsicherheit zu erreichen. Um entsprechende Erweiterungen zunächst offen zu halten, heißt das Schlüsselwort auch `type:` und nicht `class:`. Wichtig für die Robustheit und Effizienz von Programmen ist die Angabe, ob ein Feld seinen Wert ändern darf oder ob es nach seiner Initialisierung konstant bleibt, was man mit dem Schlüsselwort `constant field ...` zum Ausdruck bringt.

Eine weitere Feldannotation kann die *Allokationsart* sein. Dabei wird spezifiziert, ob das Feld eine Instanzvariable oder eine Klassenvariable (*shared slot* in CLOS) sein soll. Im letzteren Fall kann man auch noch unterscheiden, ob jede Subklasse eine eigene Speicherzelle alloziert (dies entspricht echten Klassenvariablen in ObjVLisp) oder ob sie alle dieselbe teilen. Da auf Felder aber ohnehin mit Operationsaufrufen zugegriffen wird, kann ein zu *shared slots* äquivalentes Verhalten schon mit anderen Mitteln einfacher erreicht werden, nämlich durch explizite Methodendefinitionen, z. B.

```
define class C1 (Object)
end class C1;

define constant sharedX = 99;
define method getX (obj :: C1) => (n :: Integer);
   sharedX;
end method getX;

getX(make(C1));           // ==> 99
```

Dabei werden auch die semantischen Unklarheiten von CLOS vermieden. Die Designkriterien Einfachheit und Effizienz sprechen hier deutlich gegen das Konzept von *shared slots*.

Wenn man es aber unbedingt haben möchte, dann sollte auf jeden Fall die Einschränkung gelten, daß es beim Übergang von einer Klasse zu einer Subklasse keinen Wechsel geben darf: weder von *instance slot* zu *shared slot* noch umgekehrt. Handelt es sich um echte Klassenvariablen (jede Subklasse besitzt eine eigene Speicherzelle), so ist es eher irreführend so zu tun, als ob man sie aus einer Instanz liest. Hier hat man ein deutliches Indiz dafür, daß es sich um eine Instanzvariable des Klassenobjekts handelt.

Auch bei den vorher aufgeführten Feldannotationen muß man sich fragen, ob ein konkretes Objektsystem wirklich alle unterstützen sollte. Ich komme auf diese Frage zurück, nachdem die Implementierungsaspekte im nächsten Kapitel diskutiert wurden.

4.2.2 Verhalten von Objekten

Das Verhalten von Objekten wird durch die Gesamtheit aller auf sie anwendbaren Operationen bestimmt. Es wird auch als ihr Protokoll bezeichnet. Unter dem Begriff *Operation* subsumiere ich Funktionen und Prozeduren, da es oft irrelevant ist, ob ein Ergebnis geliefert wird oder nicht.[4] An den relevanten Stellen wird dann explizit unterschieden. Dieselbe Operation kann für unterschiedliche Objektklassen unterschiedliche Realisierungen erfordern. Dabei muß man gemeinsame Verhaltensanteile als solche explizit machen können und sie genau einmal spezifizieren. Man tut dies, indem man für die jeweiligen Objektklassen entsprechende *Methoden* definiert. Methoden sind also die Bausteine von Verhaltensbeschreibungen. *Generische Operationen* fassen Methoden, die logisch verwandtes Verhalten beschreiben, zusammen. Sie dienen dazu Verhalten zu abstrahieren, weil man bei Operationsanwendung (Sendeereignis in Smalltalk) nicht genau wissen muß, welche Methode tatsächlich ausgeführt werden soll. Es reicht zu wissen, daß es eine anwendbare Methode gibt.

Operationen werden oft nicht nur auf *ein*, sondern auf mehrere Argumente angewandt. Dabei sind meistens ein oder zwei Argumente für die Auswahl der anwendbaren Methoden verantwortlich. Ein *diskriminieren* über mehr als zwei Parameter ist eher selten, soll aber nicht ausgeschlossen werden. Objektorientiertes Programmieren mit *Multimethoden* hilft, eine unnatürliche Singularisierung und somit einen Überblicksverlust zu vermeiden. Multimethoden schaffen einen fließenden Übergang von strikt komponentenorientierten zu prozeßorientierten Beschreibungen, weil nicht mehr nur ein einzelnes Objekt, sondern mehrere gleichzeitig in die Betrachtung einbezogen werden können.

Multimethoden führen schließlich auch dazu, daß generische Operationen über ihre Namen über Klassengrenzen hinweg, aber innerhalb eines Moduls, sichtbar werden. Sie werden daher auch auf der gleichen Ebene wie Klassen definiert:

```
define generic colorOf (object :: Object) => (color :: Color)
```

Mit dem Namen einer generischen Operation werden auch die Parameter und der Ergebnistyp festgelegt. Alle Methoden, die mit dem gleichen Namen definiert werden und somit

[4] In Lisp liefert jeder Ausdruck ein Ergebnis, auch wenn er ignoriert wird. Traditionell spricht man in Scheme von Prozeduren, um zum Ausdruck zu bringen, daß Operationen Seiteneffekte auslösen können, während man in allen anderen Lisp-Dialekten von Funktionen spricht, um der als Regel empfohlenen seiteneffektfreien Programmierung das Wort zu reden.

zur selben generischen Operation gehören, müssen mit ihr *kongruente Signaturen* haben, d. h.

1. die Parameterlisten müssen die gleiche Anzahl formaler Parameter haben,
2. falls die generische Operation einen *Restparameter* hat, müssen auch alle Methoden einen Restparameter haben;
3. die Parameter jeder Methode müssen vom gleichen oder spezielleren Typ sein wie die der generischen Operation,
4. der Ergebnistyp jeder Methode muß gleich oder spezieller dem der generischen Operation sein.

Restparameter erlauben optional eine beliebige Anzahl von Argumenten. Kompliziertere Parameterlisten wie in CLOS oder DYLAN können als Erweiterung bereitgestellt werden. Da es sie aber im funktionalen Teil von EuLisp nicht gibt, werden sie von Τελος nicht unmittelbar unterstützt.

Eine wichtige Einschränkung gegenüber CLOS ist die Forderung, daß es höchstens eine Methode mit einem gegebenen *Anwendungsbereich* (*domain*) geben darf. In CLOS können es mehrere sein und es hängt von der Reihenfolge der Methodendefinitionen ab, welche als letzte kommt und bleibt. Dies mag ein bequemer Weg während der Programmentwicklung sein, nicht aber eine tragbare Lösung für wiederverwendbare Software-Bausteine. Ebenso problematisch ist die Möglichkeit, Methoden zur Laufzeit durch Aufruf der Operation `remove-method` zu entfernen. Es ist nützlich während der Programmentwicklung, steht aber im Gegensatz zu den Kriterien Robustheit und Effizienz.

Folgende Methodendefinitionen wären aus meiner Sicht ein vernünftiges Beispiel:

```
define method colorOf (object :: Object) => (color :: Color);
   gray;
end method colorOf;

define method colorOf (object :: Point) => (color :: Color);
   black;
end method colorOf;
```

Dabei wird angenommen, daß die Klassen `Color` und `Point`, sowie die Instanzen von `Color` `black` und `gray` bereits definiert wurden. `Object` ist die Wurzel der Superklassenhierarchie. Dies ist übrigens ein Beispiel für statisches Verhalten, weil das Methodenergebnis überhaupt nicht vom Zustand des konkreten Arguments abhängt, sondern nur von der statisch definierten Klasse.

Methodenkombination

In praktischen Anwendungen überwiegen Methoden, die von Objektzuständen abhängige Berechnungen durchführen. Diese sind für speziellere Objetkte meistens nur anzupassen oder zu ergänzen. Um dieses abstrakt auszudrücken, braucht man ein Sprachkonstrukt,

das in einer spezielleren Methode die nächstallgemeinere ausführt. Ich nenne es ähnlich wie in CLOS `callNextMethod`. Wichtig ist jedoch, daß die nächstallgemeinere Methode auch tatsächlich auf dieselben Argumente angewandt wird, wie die aktuelle Methode. Anderenfalls kann kein robustes Verhalten vom System garantiert werden. CLOS und DYLAN, die hier großzügig sind und andere Argumente zulassen, verlangen vom Benutzer, daß für die anderen Argumente die gleichen Methoden anwendbar sein müssen. Sonst sei das Verhalten undefiniert. Dies ist jedoch mit meinen Designkriterien wie Robustheit und Effizienz (man könnte ja auch jedesmal eine Prüfung einbauen) nicht vereinbar. Und auch vom Modellierungsstandpunkt entspricht `callNextMethod` mit anderen Argumenten eher dem Delegieren einer Aufgabe an andere Objekte. Aber dann sollte man auch die generische Operation als ganzes anwenden und nicht ihre möglicherweise unvollständigen Versatzstücke über `callNextMethod`.

Mit dem einfachen Mittel `callNextMethod` lassen sich alle gutartigen Verwendungsvarianten der Standardmethodenkombination aus CLOS realisieren. So plausibel das abstraktere Konzept der Methodenkombination von CLOS für einfache Beispiele erscheint, in praktischen Anwendungen verliert man leicht die Übersicht bzw. man erlangt sie gar nicht, weil man sich das Puzzle mühsam zusammensuchen muß, wenn man versucht, Programme anderer zu verstehen. Ferner kann das in CLOS erlaubte Hinzufügen von weiteren Methoden im Client-Code die Semantik verwendeter (und abgeschlossener) Programmbausteine verändern. Somit sprechen die Designkriterien Einfachheit, Robustheit und Effizienz gegen das Methodenkombinationskonzept von CLOS. Es braucht daher vom Systemkern auch nicht unterstützt zu werden. Allerdings spricht nichts dagegen, es als Spracherweiterung mit Hilfe des Metaobjektprotokolls zu ermöglichen. In Kapitel 8.3.4 stelle ich ausgehend von CLOS ein revidiertes Konzept der Methodenkombination vor, das über `callNextMethod` hinausgeht.

Generischer Dispatch

Eines der wichtigsten Konzepte, die Objektsysteme bieten, ist der automatische *Methoden-Dispatch*. In rein prozeduralen oder auch funktionalen Sprachen muß er manuell und zwar an vielen Stellen programmiert werden. Hier definiert das Objektsystem einen Standardmechanismus. Spracherweiterungen mit Hilfe des Metaobjektprotokolls ermöglichen weitere Mechanismen, z. B. solche, die auch instanzspezifische Methoden (Eql-Methoden in CLOS bzw. Singleton-Methoden in DYLAN) unterstützen. Der systemdefinierte Methoden-Dispatch einer generischen Operation `f` mit dem Bereich $\texttt{D}_1$ ... $\texttt{D}_m$ [$\texttt{D}_{rest}$] für einen gegebenen Aufruf `f(`$\texttt{a}_1$ ... $\texttt{a}_m$ [$\texttt{a}_{m+1}$ ... $\texttt{a}_n$]`)` erfolgt in folgenden Schritten:

1. wähle die auf $a_1 \ldots a_m$ [$a_{m+1} \ldots a_n$] *anwendbaren Methoden* von `f` aus;
2. ordne die anwendbaren Methoden gemäß ihrer *Spezifizität* (die speziellste zuerst);
3. falls `callNextMethod` in einer der Methoden verwendet wird, sorge dafür, daß die entsprechende Methode gerufen werden kann;
4. wende die speziellste Methode auf die Argumente $a_1 \ldots a_m$ [$a_{m+1} \ldots a_n$] an und liefere ihr Ergebnis zurück.

Nun muß man genau definieren, wann eine Methode *anwendbar* ist und wann eine Methode *spezieller* ist als eine andere.

Eine Methode mit dem Bereich $D_1 \ldots D_m$ `[`D_{rest}`]` ist auf die Argumente $a_1 \ldots a_m$ `[`a_{m+1} `...`a_n`]` genau dann *anwendbar*, wenn die jeweilige Klasse C_i eines Arguments a_i Subklasse von D_i ist für alle $i \in (1 \ldots m)$, d. h. wenn D_i in der Klassenpräzedenzliste von C_i enthalten ist.

Eine Methode M_1 mit dem Bereich $D_{11} \ldots D_{1m}$`[`D_{rest}`]` ist *spezieller* als eine Methode M_2 mit dem Bereich $D_{21} \ldots D_{2m}$`[`D_{rest}`]` *bezüglich* der Argumente $a_1 \ldots a_m$ `[`a_{m+1} `...`a_n`]` genau dann, wenn ein $i \in (1 \ldots m)$ existiert, so daß D_{1i} in der Klassenpräzedenzliste von C_i, der Klasse von a_i, vor D_{2i} steht und für alle $j \in (1 \ldots i-1)$ D_{2j} nicht vor D_{1j} in der Klassenpräzedenzliste von C_j, der Klasse von a_j steht.

Leider ist die letzte Definition etwas kompliziert. Ihr Nachteil ist auch, daß man zwei Methoden nicht generell für alle Argumente ordnen kann. Grund dafür ist, daß man im Fall multipler Vererbung im allgemeinen keine global konsistenten Klassenpräzedenzlisten annehmen darf. Weiterer Schwachpunkt ist, daß für Multimethoden die Parameterreihenfolge herangezogen werden muß, um Mehrdeutigkeiten aufzulösen. Dies ist eine konservative Entscheidung in der Tradition von Lisp-Objektsystemen[5]. *Einfachmethoden*, also solche, die nur über einen Parameter diskriminieren, lassen sich bzgl. gegebener Argumente leicht sortieren. Im nächsten Abschnitt werde ich im Zusammenhang mit den Vererbungsstrategien Vereinfachungen diskutieren.

4.2.3 Allozieren und Initialisieren von Objekten

Oben wurde bereits gesagt, daß die definierte Struktur eines Objekts auch seine Allozierung und Initialisierung beeinflußt. So werden in der Klassendefinition die Anzahl der Felder, ggf. auch ihre Ersatzwerte und Initialisierungs-Schlüsselwörter festgelegt. Die Erzeugung eines neuen Objekts erfolgt in der Regel in zwei Schritten:

1. Es wird ein uninitialisiertes Objekt als Instanz der angegebenen Klasse alloziert.
   `allocate(C, initargs)`
2. Das neue Objekt wird initialisiert.
   `initialize(object, initargs)`

Ein systemdefinierter allgemeiner Konstruktor faßt beide Schritte zusammen:

```
        define function make (cl :: Class, #rest initargs) => (ob-
ject :: Object);
           let object = allocate(cl, initargs);
             initialize(object, initargs);
             object;
        end function make;
```

[5] In DYLAN werden Mehrdeutigkeiten nicht aufgelöst. Dort genügt es, wenn mindestens eine Methode spezieller ist als alle anderen (anwendbaren). Ein Fehler wäre aber, wenn aus so einer Methode `callNext-Method` aufgerufen würde, weil dann die Mehrdeutigkeit relevant würde.

Daneben können spezielle Konstruktoren für konkrete Klassen definiert werden. Dies kann durch eine entsprechende Option in Klassendefinitionen veranlaßt werden.

```
define class Point (Object)
  field x initKeyword: x:, initValue: 1, reader: pointX;
  field y requiredInitKeyword: y:, reader: pointY;
  constructor makePoint (x:, y:);
end class Point;

pointX(makePoint(11, 22));        // ==> 11
pointY(makePoint(11, 22));        // ==> 22
```

Die *Klassenannotation* `constructor` bewirkt im obigen Beispiel, daß automatisch ein Konstruktor im Prinzip wie folgt definiert wird:

```
define function makePoint (x, y) => (object :: Point);
  let initargs = list(x:, x, y:, y);
  let object = allocate(Point, initargs);
    initialize(object, initargs);
    object;
end function makePoint;
```

Während der Diskussion der Implementierungstechniken wird hierzu eine effizientere Lösung entwickelt.

Beide Operationen, `allocate` und `initialize`, sollten dem Benutzer direkt zur Verfügung stehen. Somit kann er auch selbst einen speziellen Konstruktor für eine gewünschte Klasse definieren oder die Einzelschritte in Eigenregie durchführen, also z. B. die Initialisierung zu einem späteren Zeitpunkt durchführen. Spezialisieren darf er aber zunächst nur `initialize`, d. h. neue Methoden definieren. Benutzerdefinierte `allocate` Methoden bleibt der Metaobjektebene vorbehalten (siehe Abschnitt 4.4.6).

Diese Lösung ist im Prinzip ähnlich wie in CLOS. Sie unterscheidet sich von Java darin, daß dort die Initialisierungsmethoden als Konstruktoren bezeichnet werden und `new` ein syntaktisches Schlüsselwort ist, keine reguläre Operation wie `make` hier. Es gibt dort zwar auch eine allgemeine Konstruktor-Operation `newInstance`, ihr kann man aber keine Intitialisierungsargumente mitgeben. In Java kann man auch nicht verhindern, daß alle anwendbaren Initialisierungsmethoden[6] automatisch ausgeführt werden. Aus meiner Sicht soll der Benutzer entscheiden können, ob die allgemeineren `initialize` Methoden über `callNextMethod` zum Zuge kommen sollen oder nicht.

Ferner sollen optionale Initialisierungsargumente unterstützt werden, wie auch in CLOS, und im Unterschied zu Java, wo man für jeden potentiellen Fall einen speziellen Konstruktor (d. h. eine Initialisierungsmethode) mit geforderten Parametern spezifizieren muß. Die Java-Lösung zieht es nach sich, daß das Initialisierungsverhalten selbst innerhalb einer Klasse auf mehrere Konstruktoren aufgeteilt ist, was das Programmverstehen und die Programmwartung erschwert.

Die Initialisierung von Objekten erfolgt in der systemdefinierten *initialize* Methode in zwei Schritten:

[6] Dies sind alle Konstruktoren mit der gleichen Parameterliste.

1. Überprüfe die *Zulässigkeit der Initialisierungsargumente* anhand der Initialisierungsschlüsselwörter der Klasse.[7]

2. Initialisiere jedes Feld eines Objekts falls vorhanden gemäß seinem Initialisierungsargument, sonst gemäß seinem Ersatzwert; fehlt ein gefordertes Initialisierungsargument, melde einen Fehler.

Die *Zulässigkeit der Initialisierungsargumente* ist durch die direkten und geerbten Feldannotationen `initKeyword:` und `requiredInitKeyword:` bestimmt. Es können auch noch zusätzliche Initialisierungsschlüsselwörter als Klassenannotation hinzukommen, so daß entsprechende Initialisierungsargumente dann in selbstdefinierten `initialize` Methoden behandelt werden können.

Einige Felder können zunächst uninitialisiert bleiben. Sie müssen jedoch vor ihrem ersten Lesezugriff einen Wert erhalten.

4.2.4 Lesen und Schreiben von Feldinhalten

Wie schon bei den Ausführungen zur Struktur von Objekten gesagt, sollen Feldzugriffe aus Uniformitätsgründen wie reguläre Operationsaufrufe notiert werden. Wegen Multimethoden kommt die Notation von Feldzugriffen wie auf lokale Variablen (Smalltalk, Flavors) nicht in Frage, da nicht eindeutig wäre, welches Objekt (Parameter) gemeint sei. Es soll keine anderen Primitive für Feldzugriffe geben, als die in Klassendefinitionen spezifizierten. Will man möglichst schnelle Feldzugriffe, so dürfen sie nicht generisch sein. Das zieht eine Beschränkung der Funktionalität nach sich, z. B. nur einfache Vererbung, keine explizite Methodendefinitionen für Lese- und Schreiboperationen usw.

Auf der Metaobjektebene gibt es dann ein Protokoll, wie die Lese- und Schreiboperationen generiert werden und wie dieses Verhalten vom Benutzer spezialisiert werden kann (siehe Abschnitt 4.4.5 auf Seite 104).

4.3 Spezialisieren und Generalisieren von Objektklassen

In diesem Abschnitt werden die wichtigsten Vererbungsstrategien im Kontext des Entwurfs einer objektorientierten Programmiersprache diskutiert. Ein weitgehend klares Konzept stellt die *einfache Vererbung* dar. Hierfür stehen ausgereifte und effiziente Implementierungstechniken zur Verfügung. Nachteilig wirkt sich die Beschränkung auf höchstens eine Superklasse dann aus, wenn es mehrere *orthogonale Generalisierungskriterien* gibt, also z. B. wenn man Fahrzeuge einerseits nach ihren Bestandteilen kategorisiert (partonomisch) und andererseits nach den Verwendungsmöglichkeiten (funktional). Es kommt dann zwangsläufig zu redundantem Code und der Wiederverwendungsgrad von Softwarebausteinen sinkt. *Multiple Vererbung* unterstützt modulares Generalisieren bzgl. mehrerer orthogonaler Kriterien, was auch den Wiederverwendungsgrad erhöht. Ohne weitere Beschränkungen können jedoch die Vorteile multipler Vererbung auch ins Gegenteil umschlagen. Beispielsweise dann, wenn man eben nicht auf die Orthogonalität bei mehreren

[7] Dieser Schritt kann in weniger dynamischen Sprachen immer, und hier im Rahmen von Optimierungen in vielen Fällen, in die Übersetzungsphase verlagert werden.

direkten Superklassen achtet und es zwangsläufig zu Konflikten[8] und Mehrdeutigkeiten bei der Vererbung kommt. Ihre implizite Auflösung durch Linearisierungsverfahren der Vererbungshierarchie sind nicht immer intuitiv oder gar transparent und vorhersehbar für den Programmierer [Bretthauer *et al.*, 1989c]. Explizite Auflösung von Konflikten und Mehrdeutigkeiten durch Umbenennen oder Auswahl durch den Programmierer bricht die Abstraktion und führt zwangsläufig zu weniger generischen und schlechter verfeinerbaren Bausteinen [Carre und Geib, 1990]. Entsprechend aufwendiger sind auch die Implementierungstechniken und weniger effizient die Laufzeitsysteme bzw. die Übersetzungsresultate bei multipler Vererbung.

Bemerkenswert ist auch die nahezu weltanschauliche Spaltung der Gemüter, wenn es um die Frage multipler vs. einfacher Vererbung geht. Da dieser Religionsstreit noch nicht entschieden [Queinnec, 1996, S. 418] ist, beziehe ich als Sprachdesigner eine pragmatische Position. Was einfach zu lösen ist, sollte auch mit einfachen Mitteln gelöst werden. An performanzkritischen Stellen, wie den systemdefinierten Klassen und Operationen, sollte die Laufzeiteffizienz ein höheres Gewicht erhalten als die Vermeidung von Coderedundanz. Das heißt, der Objektsystem-Kern sollte mit Hilfe einfacher Vererbung organisiert sein. Als Standardfunktionalität sollte vom System ebenfalls einfache Vererbung angeboten werden. Es sollten aber auch systemdefinierte Erweiterungsmodule für multiple und Mixin-Vererbung bereitgestellt werden. Darüberhinaus soll es dem Benutzer ermöglicht werden, eigene Vererbungskonzepte als Spracherweiterung zu realisieren und zu benutzen.

Nun präsentiere ich die drei wichtigsten Vererbungskonzepte unter den Gesichtspunkten des Spezialisierens und Generalisierens von Struktur und Verhalten von Objektklassen. Zunächst folgen noch einige Definitionen, die allen Vererbungskonzepten gemeinsam sind.

Als *direkte Superklassen* einer Klasse werden solche bezeichnet, die in ihrer Definition explizit als Superklassen spezifiziert wurden oder, falls die Angabe von Superklassen optional ist, die angenommene Default-Superklasse `Object`. Jede Klasse bis auf `Object` hat mindestens eine direkte Superklasse. Sei $\mathcal{K}$ die Menge aller definierten Klassen. Die Relation $ds := \{(k_1, k_2) \mid k_2 \text{ ist direkte Superklasse von } k_1\}$ spannt die Klassenhierarchie auf. Von ds verlange ich, daß sie irreflexiv ist. Auch die transitive Hülle von ds, $\tilde{ds}$, sei irreflexiv. Daraus folgt auch schon, daß der induzierte *Vererbungsgraph* $G = (\mathcal{K}, ds)$ azyklisch ist.

4.3.1 Einfache Vererbung

Einfache Vererbung liegt dann vor, wenn eine Klasse höchstens eine direkte Superklasse besitzt. Da oben auch schon festgelegt wurde, daß sie auch mindestens eine direkte Superklasse haben soll, hat die Klassenhierarchie die Form eines Baumes $B = (\mathcal{K}, ds^{-1})$ mit der Wurzel `Object`. Die *Klassenpräzedenzliste cpl* für eine Klasse k ergibt sich als Liste der Klassen, die auf dem Pfad von k zu `Object` liegen: $cpl = (k, \ldots Object)$.

Ob eine Klasse k_1 spezieller ist als eine andere Klasse k_2, läßt sich leicht und intuitiv bestimmen: (k_1, k_2) muß in $\tilde{ds}$ enthalten sein bzw. k_2 muß in der Klassenpräzedenzliste von k_1 enthalten sein. Alle Klassenpräzedenzlisten sind *global konsistent*, d. h. wenn k_1 in einer Klassenpräzedenzliste vor k_2 kommt, gilt diese Reihenfolge auch in allen anderen

[8] Hier sind nicht unbeabsichtigte Namenskonflikte gemeint. Solche werden vom Modulkonzept erkannt und ggf. durch Umbenennung gelöst.

Klassenpräzedenzlisten, in denen k_1 und k_2 gemeinsam vorkommen. Diese Eigenschaft wird auch als *Monotonie* bezeichnet [Ducournau und Habib, 1991].

Die Berechnung der Klassenpräzedenzliste *cpl* ist auch sehr einfach und kann rekursiv wie folgt definiert werden:[9]

```
define function computeCPL (k :: Class) => (l :: List) ;
  if k == Object
   then list(Object);
   else cons(k, computeCPL(ds(k)));  // ds(k) liefert die direkte
  end if;                            // Superklasse von k
end function computeCPL;
```

Spezialisieren und Generalisieren von Struktur

Wird eine Klasse als direkte Subklasse einer anderen Klasse definiert, so *erbt* sie alle Felder der Superklasse. Da Vererbung zur Spezialisierung verwendet wird, macht eine selektive Vererbung keinen Sinn. Würden der Subklasse Felder der Superklasse fehlen, könnte sie auch nicht als ihre Spezialisierung angesehen werden. Insofern ist die Sprechweise einiger Programmiesprachen, private Felder würden nicht vererbt, irreführend. Man kann höchstens ihre Sichtbarkeit beschränken.

Ein Feld zu spezialisieren bedeutet vor allem, daß sein Wert von einem spezielleren Typ sein soll und daß der Ersatzwert entsprechend spezieller sein muß als in der Superklasse. Dies betrifft also die Feldannotationen `type:`, `initValue:`, `initFunction:` und `initExpression:`. Dabei sollen die generellen Methoden der Superklasse mit dem spezielleren Feld problemlos umgehen können. In C++ und Java ist dies leider nicht der Fall. Speziellere Felder mit dem gleichen Namen führen zu Feldduplikaten, die nur aus spezielleren Methoden referiert werden können.

Darüber hinaus könnten auch die Annotationen `accessor:`, `reader:` und `writer:` in einer Subklasse für ein ererbtes Feld spezifiziert werden. Im Fall der Zugriffsoperationen halte ich es jedoch für sinnvoll, an genau einer Lese- und genau einer Schreiboperation für ein Feld festzuhalten. Werden diese Annotationen mehrfach spezifiziert, so sollen keine neuen Operationen generiert werden, sondern die einmal vorhandenen an weitere Bezeichner gebunden werden. Dies ist anders als in CLOS, wo für jeden neuen Bezeichner auch eine neue generische Operation generiert wird.

Die Annotationen `initKeyword:` und `requiredInitKeyword:` könnten ebenso für ererbte Felder spezifiziert werden. In CLOS kann es auf diese Weise zu mehreren Initialisierungsschlüsselwörtern für ein Feld kommen. Dies verlangsamt jedoch die Initialisierung von Objekten. Auch hier kann dasselbe Prinzip wie bei Zugriffsoperationen angewandt werden, um eine konzeptuell klare und effiziente Lösung zu erhalten. Sie werden als Bezeichner aufgefaßt, die für ein und dasselbe Feld alle an denselben Wert gebunden sind. Für die interne Verarbeitung spielt nur dieser Wert eine Rolle. Da es pro Feld höchstens einer ist, kann die Implementierung einfach und effizient sein. Da es mehrere Bezeichner geben darf, fällt es dem Benutzer leicht geeignete Abstraktionen auszudrücken.

[9] Natürlich wird man in der realen Implementierung eine endrekursive Variante wählen.

Wenn Felder der Superklasse spezialisiert werden, so muß der Programmierer dies explizit machen, wie man oben schon gesehen hat:

```
define class myPoint (Point)
  specialize field x initValue: 37;
  field z;
  ...
end class myPoint;
```

Das heißt nicht, daß man alle Felder der Superklasse kennen muß. Man sollte nur diejenigen kennen, die man spezialisieren will. Sie müssen natürlich auch sichtbar sein. Die Sichtbarkeit von Feldnamen bedeutet noch nicht, daß man auf einen entsprechenden Wert in einem Objekt zugreifen kann. Dafür müssen die Zugriffsoperationen sichtbar sein.

In CLOS ist dies anders. Dort kann es daher leichter zu Konflikten bzw. unbeabsichtigter Spezialisierung von Feldern der Superklasse kommen. Objektkapselung kann durch die Kenntnis von Slotnamen durchbrochen werden, indem man die Operation `slot-value` aufruft.

Zusammenfassend kann man festhalten, wie die Annotationen für ein Feld f der Klasse k_1 in der Subklasse k_2 spezialisiert werden, und zwar auf die explizite Direktive `specialize` hin:

- ist eine Annotation in k_1 unspezifiziert[10], so gilt die Angabe in k_2;
- für die Annotationen `type:`, `initValue:`, `initFunction:`, `initExpresseion:` gilt die Angabe in k_2 (*Overriding*); für die Typangabe muß aber gelten, daß der Typ dem in k_1 gleich oder spezieller ist;
- für die Annotationen `reader:`, `writer:`, `accessor:`, `initKeyword:`, `requiredInitKeyword:` gilt die *Vereinigung* aller Angaben aus k_1[11] und k_2. Hierbei ist zu beachten, daß es für ein Feld höchstens ein Initialisierungsschlüsselwort-Objekt, höchstens eine Lese- und höchstens eine Schreiboperation gibt. Die Vereinigung bezieht sich auf die Bezeichner, von denen es beliebig viele für ein und dieselbe Annotation eines Feldes geben kann. Die Bezeichner haben somit eine rein syntaktische Funktion, die aber wichtig für die Verständlichkeit von Programmen ist.

Ein wesentlicher Vorteil einfacher Vererbung ist ihr lokaler Charakter. Wurde für eine Klasse k_1 die Vererbung durchgeführt, so braucht man für ihre Subklassen nichts mehr über die Superklassen von k_1 zu wissen. Für die Subklassen von k_1 ist es auch irrelevant, ob bestimmte Annotationen direkt in k_1 spezifiziert oder ob sie von den Superklassen von k_1 geerbt wurden.

Spezialisieren und Generalisieren von Verhalten

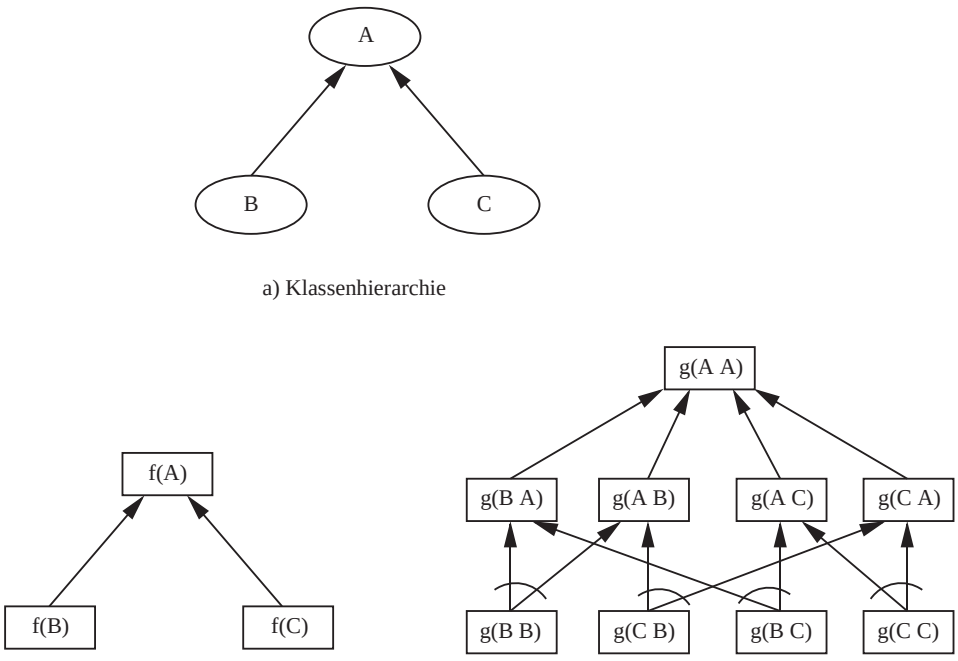
Zum Spezialisieren und Generalisieren von Verhalten ist bereits alles wesentliche oben erwähnt worden. Die entscheidenden Merkmale sind der *Methoden-Dispatch* (potentiell zur

[10] Das heißt, sie wurde weder in k_1 noch in einer der Superklassen von k_1 spezifiziert.
[11] Das heißt, der für k_1 direkt spezifizierten und der von Superklassen von k_1 geerbten Angaben.

Laufzeit) sowie das Konstrukt `callNextMethod`. Letzteres ist im Unterschied zu CLOS und DYLAN parameterlos, um sicherzustellen, daß die nächstallgemeinere Methode auch tatsächlich auf dieselben Argumente angewandt wird wie die aktuelle Methode. Die Beschränkung auf einfache Vererbung ließe aber eine einfachere Beschreibung des Methoden-Dispatches zu, als sie oben gegeben wurde. Insbesondere kann bei der Bestimmung für Einfachmethoden, welche der anwendbaren Methoden spezieller ist, der Bezug auf aktuelle Argumente eines Aufrufs entfallen. In Programmen, die keine neuen Methoden zur Laufzeit erzeugen, wäre für Einfachmethoden sogar statisch entscheidbar, welche Methode mit einem Aufruf von `callNextMethod` gemeint ist, ohne daß der Programmierer die Abstraktion brechen muß. Leider ist dies für Multimethoden im allgemeinen nicht möglich.

Abbildung 4.1 verdeutlicht den Zusammenhang zwischen einfacher Vererbung und dem Methoden-Dispatch bei Einfach- und Multimethoden. Sei A die direkte Superklasse von B und C, und seien $f(A)$, $f(B)$ und $f(C)$ Einfachmethoden. Dann ist statisch entscheidbar, welche Methode spezieller ist als eine andere. Ein `callNextMethod` in $f(B)$ oder in $f(C)$ würde $f(A)$ referieren. Vergleichbare Methoden bilden eine lineare Ordnung.

Für die Multimethoden $g(AA)$, $\ldots g(CC)$ erhält man ohne Berücksichtigung der Parameterreihenfolge nur noch eine partielle Ordnung, so daß ein `callNextMethod` in $g(BB)$ zwischen $g(BA)$ oder $g(AB)$ auswählen müßte. Ein `callNextMethod` in $g(BA)$ deutet zunächst auf $g(AA)$. Falls aber der ursprüngliche Aufruf $g(bc)$ lautete (mit b direkte Instanz von B und c direkte Instanz von C), so müßte vor $g(AA)$ noch die Methode $g(AC)$ berücksichtigt werden, die spezieller ist. In der Abbildung 4.1 wurden jeweils alle möglichen Methoden für die gegebenen Klassen A, B und C aufgeführt.



a) Klassenhierarchie

b) Einfachmethoden

c) Ambiguität bei Multimethoden

Abbildung 4.1: Einfache Vererbung und Methoden-Dispatch

Ausgehend von den Designkriterien in Kapitel 2 wäre es wünschenswert, eine einfache und intuitive Einschränkung von Methodenkonstellationen zu finden, die den Rückgriff auf die Parameterreihenfolge überflüssig macht. Solange dies nicht gelungen ist, bleibe ich bei der für Lisp traditionellen Lösung und sortiere Methoden gemäß der Spezifizität ihrer Parameter von links nach rechts, d. h. $g(BA)$ kommt *vor* $g(AB)$.

4.3.2 Multiple Vererbung

Multiple Vererbung liegt dann vor, wenn eine Klasse mehr als eine direkte Superklasse besitzt. Multiple Vererbung ist notwendig, wenn man bzgl. mehrerer Kriterien generalisieren möchte. Dann trägt sie zu besseren Abstraktionen in Programmen bei. Das Klassifizieren wird modularer. Schwierigkeiten treten dann auf, wenn die Kriterien nicht orthogonal sind und als Folge unbeabsichtigte bzw. unerwünschte Effekte auftreten, weil z. B. das gleiche Feld in mehreren direkten Superklassen definiert ist oder für dieselbe generische Operation nun Methoden geerbt werden sollen.

Zunächst soll auch für multiple Vererbung gelten, daß eine Klasse mindestens eine direkte Superklasse haben soll. Da zyklische Abhängigkeiten ausgeschlossen werden, bekommt die Klassenhierarchie die Form eines gerichteten azyklischen Graphen (DAG) $G = (\mathcal{K}, ds^{-1})$ mit dem kleinsten Knoten `Object`. Die *Klassenpräzedenzliste cpl* für eine Klasse k ergibt sich als Linearisierung einer partiellen Ordnung, beginnend mit k und `Object` am Ende.

Ob eine Klasse k_1 spezieller ist als eine andere Klasse k_2, läßt sich nicht mehr leicht und intuitiv bestimmen. Die Klassenpräzedenzlisten sind per Konstruktion weder *global konsistent* noch *monoton* [Ducournau und Habib, 1991]. Diese Eigenschaften lassen sich für beliebige Klassenhierarchien auch nicht durch geschickte Linearisierungsverfahren herstellen [Bretthauer *et al.*, 1989c]. Um all diesen Problemen aus dem Weg zu gehen und nicht von der Reihenfolge direkter Superklassen abhängig zu sein, wird in C++ eine Linearisierung erst gar nicht versucht. Im Konfliktfall wird ein Übersetzungsfehler gemeldet. Der Programmierer muß dann z. B. bei Operationsaufrufen (bei *virtual functions*) explizit angeben, welche Methode genommen werden soll. Dies bricht jedoch die Abstraktion, so daß Programme weniger generisch und schwerer wartbar werden [Carre und Geib, 1990].

Die Berechnung der Klassenpräzedenzliste *cpl* kann nur dann nachvollziehbar und intuitiv spezifiziert werden, wenn man problematische Hierarchiearten ausschließt. Auch diese Einschränkung muß natürlich möglichst einfach und plausibel sein. In CLOS werden nur solche Hierarchien ausgeschlossen, in denen die direkten Superklassen einer zu definierenden Klassen jeweils umgekehrte Reihenfolgen ihrer direkter Superklassen aufweisen. Monotonie oder gar globale Konsistenz der Klassenpräzedenzlisten wird dadurch noch nicht sichergestellt.

In DYLAN sind zwei Klassen k_1 und k_2 zwangsweise disjunkt, wenn es in ihren Klassenpräzedenzlisten Elemente k_i und k_j gibt, die umgekehrt geordnet sind. Das heißt, daß keine neue Klasse von beiden Klassen, k_1 *und* k_2, direkt oder indirekt erben darf. Auf diese Weise wird zwar die Monotonie der Klassenpräzedenzlisten als eine Art lokaler Konsistenz sichergestellt, nicht aber die globale Konsistenz aller Klassenpräzedenzlisten.

In OZ ist eine Klassendefinition *nicht zulässig*, wenn für irgendein Feld oder irgendeine Methode die induzierte partielle Ordnung aus der Superklassenrelation und den jeweiligen Reihenfolgen direkter Superklassen in Klassendefinitionen kein kleinstes (speziellstes)

Element existiert. Diese Lösung hat den Nachteil, daß erst auf der untersten Granularitätsstufe entschieden werden kann, ob eine Klassendefinition zulässig ist. Sie ist auch sehr empfindlich, was Programmänderungen angeht. Als Programmierer muß man den gesamten Kontext kennen, um z. B. vorherzusehen, ob durch Hinzufügen oder Entfernen einer Methode, einige Klassen unzulässig werden.

Bevor ich hierzu meine Designentscheidung treffe, sollen die Auswirkungen multipler Vererbung auf Struktur und Verhalten diskutiert werden.

Spezialisieren und Generalisieren von Struktur

Als erstes Problem taucht bei multipler Vererbung die Frage auf, wie man mit Feldern gleichen Namens aus mehrerern direkten Superklassen umgeht. Angenommen, es wird eine Klasse A als direkte Subklasse von B und C definiert:

```
define class A (B, C)
end class A;
```

Hier kann es zur Spezialisierung von Feldannotationen kommen, auch ohne daß dies explizit in der Definition von A veranlaßt wird. Dies trifft dann zu, wenn B und C ein Feld f von einer gemeinsamen Superklasse D geerbt haben ("Diamant"-Struktur). Falls auf keinem der Vererbungspfade von D nach B und von D nach C Spezialisierungen vorgenommen wurden, könnte A dieses Feld unverändert durchgereicht bekommen. Anderenfalls muß man festlegen, wie die Feldannotationen von f in A aus denen in B und C zusammengesetzt werden sollen. Dabei könnte man annehmen bzw. festlegen, daß direkte Angaben in B spezieller als diejenigen in C seien. Dies wäre aber nur ein Spezialfall. Im allgemeinen kann bei multipler Vererbung die Betrachtung der gesamten Vererbungshierarchie ausgehend von A notwendig werden. Will man die Betrachtung auf B und C beschränken dürfen (Lokalitätseigenschaft), so müssen Restriktionen über deren Superklassenhierarchien angenommen werden. Somit stellt sich wieder die Entscheidungsfrage nach der Linearisierung der Vererbungshierarchie.

Um die Designentscheidungen möglichst modular treffen zu können, nehme ich hier einfach an, daß es eine Linearisierung in Form der Klassenpräzedenzliste gibt. Wie man zu einer adäquaten Linearisierung kommt, behandele ich später im Abschnitt 4.2. Zunächst habe ich das Problem reduziert, um es handhabbar zu machen. Das Ergebnis stellt demzufolge zunächst auch noch keine vollständige Lösung dar. Ich stütze die Beschreibung der Vererbung von Struktur auf die Klassenpräzedenzliste. Dadurch werden mögliche Spracherweiterungen besser vorbereitet. Selbst wenn jemand eine eigene Berechnungsmethode der Klassenpräzedenzliste definiert, sorgt das System für eine vernünftige Vererbung der Felder, ohne daß diese Basismethode auch neu gemacht werden müßte.

Im Prinzip ist der Algorithmus ähnlich wie bei einfacher Vererbung, nur reichte es dort aus, genau eine (direkte) Superklassen zu betrachten. Hier muß man alle Superklassen (direkte und indirekte) in der Reihenfolge der Klassenpräzedenzliste betrachten.

Sei die Klassenpräzedenzliste mit $(k_1, \ldots k_n)$ gegeben. Dann werden die Annotationen für ein Feld f wie folgt vererbt:

- ist eine Annotation in keiner der Klassen $k_1, \ldots k_n$ spezifiziert, so bleibt sie unspezifiziert;
- für die Annotationen `initValue:`, `initFunction:`, `initExpression:` gilt die speziellste Angabe, also die in k_i, so daß in den Klassen $k_1, \ldots k_{i-1}$ keine dieser Annotationen spezifiziert wurde (*Overriding*);
- für die Annotation `type:` gilt ebenfalls *Overriding*, allerdings muß für die gemäß der Klassenpräzedenzliste speziellste Angabe T in k_i gelten, daß T Subtyp aller anderen Typangaben in $k_{i+1}, \ldots k_n$ ist. Anderenfalls müßte entweder ein Fehler gemeldet werden oder es müßte ein neuer Typ als Durchschnitt aller Typangaben in $k_i, \ldots k_n$ vom System gesucht bzw. automatisch erzeugt werde.
- für die Annotationen `reader:`, `writer:`, `accessor:`, `initKeyword:`, `requiredInitKeyword:` gilt die *Vereinigung* aller Angaben aus $k_1 \ldots k_n$. Hier ändert sich nichts im Vergleich zur einfachen Vererbung.

Für die Annotation `type:` würde man eigentlich den mengentheoretischen Durchschnitt aller Angaben zugrundelegen wollen, d. h. daß die Werte für f in Instanzen von k_1 alle Typangaben in $k_1, \ldots k_n$ erfüllen. Eine solche Klasse, die diesen Durchschnitt induziert, wird aber möglicherweise im Programm gar nicht explizit definiert. Es stellt sich daher die Frage, ob so eine Klasse bzw. Typ implizit vom System erzeugt werden soll und wie die Einhaltung so einer Typrestriktion erfolgen soll. Will man zur Übersetzungszeit die Korrektheit von Feldzuweisungen zusichern (*compile time type checking*), so müssen auch mindestens die für die Typprüfungen benötigten Klassen zur Übersetzungszeit bekannt sein. Es kann aber auch gültige (bzgl. Typkorrektheit) Programme geben, die erst zur Laufzeit entsprechende Klassen sowie ihre Instanzen erzeugen.

Die Typannotation in Verbindung mit multipler Vererbung zum Zweck der Spezialisierung von Struktur hat also sehr weitreichende Konsequenzen auf eine Programmiersprache, will man es *richtig* machen[12]. Im Kontext einer dynamischen Sprache wie Lisp (ohne strenge, zur Übersetzungszeit überprüfbare Typisierung) könnte man akzeptieren, wenn Typfehler erst zur Laufzeit festgestellt werden, zumal die Erzeugung von Klassen zur Laufzeit nicht grundsätzlich ausgeschlossen wird. Nicht akzeptabel ist meiner Meinung nach die CLOS-Lösung, wonach das Programmverhalten undefiniert ist, wenn Typrestriktionen vom Programmierer nicht sichergestellt werden. Die von mir gewählte Vorgehensweise stellt einen Kompromiß zwischen der gewünschten Flexibilität und Robustheit dar.

Spezialisieren und Generalisieren von Verhalten

Für das Spezialisieren und Generalisieren des Verhaltens von Objekten bringt multiple Vererbung weitere Mehrdeutigkeiten. Konnte bei einfacher Vererbung wenigstens für Einfachmethoden eine statische Auflösung von `callNextMethod` Aufrufen gegeben werden, so ist dies bei multipler Vererbung im allgemeinen nicht mehr möglich. Gäbe es für die Klassen B und C jeweils die Methoden m_B und m_C:

[12] Es verwundert daher nicht, daß Sprachen wie C++ und Java die Spezialisierung von Feldannotationen überhaupt nicht unterstützen. Felder werden dort unverändert geerbt. Braucht man etwas spezielleres, muß ein neues Feld angelegt werden.

```
    define method m (x :: B);
      ...
      callNextMethod();
      ...
end method m;

    define method m (x :: C);
      ...
      callNextMethod();
      ...
end method m;
```

so kann erst im Kontext eines konkreten Aufrufs von m, z. B. für eine Instanz von A, entschieden werden, daß `callNextMethod` m_B zur Ausführung der Methode m_C führt. Als Programmierrichtlinie sollte man natürlich versuchen, in Programmen so wenig Reihenfolgeabhängigkeiten wie möglich zu haben. Dennoch kann man sie nicht völlig vermeiden. Daher ist die Forderung nach Monotonie der Klassenpräzedenzlisten für das Verhalten von Objekten besonders wichtig. Nur so können Methoden verläßlich spezialisiert und generalisiert werden. Die Reihenfolge (m_B, m_C) für A darf für Subklassen von A nicht umgekehrt werden. In CLOS ist dies nicht garantiert [Bretthauer *et al.*, 1989c].

Die Suche nach eingeschränkter multipler Vererbung mit den gewünschten Eigenschaften wie Monotonie, globaler Konsistenz und Lokalität führt schließlich zur *Mixin-Vererbung*, die im nächsten Unterabschnitt beschrieben wird.

4.3.3 Mixin-Vererbung

Die Begriffe *Mixin* und *Flavor* entstanden am MIT als Metaphern aus Steves Eisdiele in Cambridge, Massachussetts, wo auf Wunsch Nuß- und Schokoladenstücke in eine Portion Vanilleeis hineingemischt (Mixin) wurden, um ein Eis mit individuellem Geschmack (Flavor) zu bekommen. Im Kontext objektorientierter Programmierung bedeuten diese Metaphern, daß man neue Klassen als Mixtur[13] aus vorhandenen definiert, wobei einige davon eine Art Grundlage bilden, die mit verschiedenen Ingredienzien angereichert werden. Entwirft man Klassenbibliotheken nach dieser Metapher, so entstehen in der Regel flachere Klassenhierarchien mit klareren Beschreibungen, sei es als Dokumentation, welche Klassen zusammengemischt werden können und welche nicht. Es müssen auch nicht alle Kombinationen von Klassen einer Klassenhierarchien im voraus definiert werden. Vielmehr geschieht dies durch den Benutzer nach Bedarf, der die Bausteine wiederverwendet ohne sie reimplementieren zu müssen. Oft ist dies ohne weitere Verfeinerungen (Spezialisierungen von Feldern und Methoden) möglich.

[13] Ich verwende hier bewußt nicht den Begriff Komposition, um die irreführende Vorstellung zu vermeiden, im Ganzen seien die Bestandteile bzw. Ingredienzien noch als etwas eigenständiges identifizierbar. Aus dieser Vorstellung leiten sich dann solche syntaktische Konfusionen ab, daß man analog zu `this.display()` plötzlich `super.display()` notieren muß, als würde ein anderes Objekt angesprochen; dabei soll nicht ein anderes Objekt, sondern die nächstallgemeinere Methode auf demselben Objekt ausgeführt werden, also wäre z. B. die Notation `this.nextMethod()` viel klarer. Das Gegenteil soll assoziiert werden: das Ergebnis ist *ein* Ding mit verschiedenen Eigenschaften.

In FLAVORS wurden Mixins als Namenskonvention verwendet. Es gab dafür keine syntaktische oder semantische Unterstützung seitens des Objektsystems. Dies wurde nach meiner Kenntnis erstmals in [Bretthauer *et al.*, 1989c] gefordert und in MCS realisiert [Bretthauer und Kopp, 1991]. Bracha und Cook [Bracha und Cook, 1990] haben die Metapher etwas verkürzt und ein Vererbungskonzept entwickelt, bei dem es nur noch Mixins (ohne Flavors) gibt, die flach zusammengesetzt werden. Auf der Strecke bleibt dabei das Konzept des Spezialisierens und Generalisierens von Objektklassen. Übrig bleibt die explizit lineare Komposition von Programmbausteinen mit der Möglichkeit der Bausteinduplizierung, falls ein Mixin explizit oder implizit mehrfach vorkommt. Komposition paßt aber viel eher zur Modularisierung als zur Klassifikation, und so kommt Bracha auch zur Sicht der Mixins im Sinne von Modulen [Bracha, 1992] bzw. der Vererbung im Sinne der Modularisierung [Bracha und Lindstrom, 1992]. Diese Einschätzung wird auch in [Flatt *et al.*, 1998] vertreten, wo ein spezielles Konzept der Mixins für Java vorgeschlagen wird. Aber auch hier stehen die Aspekte des Spezialisierens und Generalisierens von Objektklassen sowie ihre effiziente Implementierung nicht im Vordergrund. Insbesondere dürfte das sogenannte *View-Konstrukt* mit Effizienzproblemen verbunden sein.

Das in dieser Arbeit favorisierte Mixin-Konzept greift die ursprüngliche Metapher wieder auf. Sie soll helfen, gezielt und auf systematische Weise klare und erweiterbare Klassenhierarchien zu entwerfen. Der bereitzustellende Mechanismus soll die Vorteile einfacher und multipler Vererbung verbinden sowie deren Nachteile möglichst vermeiden. Insbesondere soll er effizient implementierbar sein.

Es sollen zwei Arten von Klassen unterschieden werden. Sogenannte *Basisklassen* repräsentieren Dinge und werden umgangssprachlich mit Substantiven benannt, z. B. `window`. *Mixin-Klassen* explizieren aditive Eigenschaften von Dingen und werden mit Adjektiven benannt, z. B. `bordered`, `titled` etc. Auf diese Weise erhält man eine Richtlinie für den Aufbau einer Ontologie einer Domäne. Betrachtet man die Dinge allein, so werden diese nach dem Prinizip einfacher Vererbung klassifiziert. Eine Basisklasse kann also von höchstens einer Basisklasse erben (bzw. sie spezialisieren)[14]. Mixin-Klassen können nach dem Prinzip multipler Vererbung klassifiziert werden. Ähnlich wie man in der natürlichen Sprache mehrere Adjektive einem Substantiv voranstellen darf, können auch Basisklassen von mehreren Mixin-Klassen erben.

Dieses Konzept der multiplen Vererbung ist im wesentlichen auch für DYLAN übernommen worden, wenn auch unter anderem Namen. Basisklassen werden dort als Primärklassen (engl. *primary*), Mixin-Klassen als freie Klassen (engl. *free*) bezeichnet [Shalit, 1996, S. 134]. Genau wie hier, bilden Primärklassen in DYLAN eine einfache Vererbungshierarchie.

Abbildung 4.2 zeigt an einem in der Literatur immer wieder aufgeführtes Beispiel [Kopp, 1996a, S. 28] die Unterschiede in der Modellierung mit allgemeiner multipler und Mixin-Vererbung. Die explizite Definition der beiden Mixin-Klassen `framed` und `titled` wird auch als *Faktorisierung* bezeichnet und führt zu einer modularen Generalisierung der Funktionalität.

[14] Diese Einschränkung gilt auch weitgehend für Begriffe der natürlichen Sprache, also z. B. Deutsch. In zusammengesetzten Substantiven hat fast immer nur ein Teilsubstantiv die Rolle eines Dings, während alle anderen, meist vorangestellten, Teilsubstantive beschreibende Funktion haben, z. B. Eisbecher, Vanilleeis, etc.

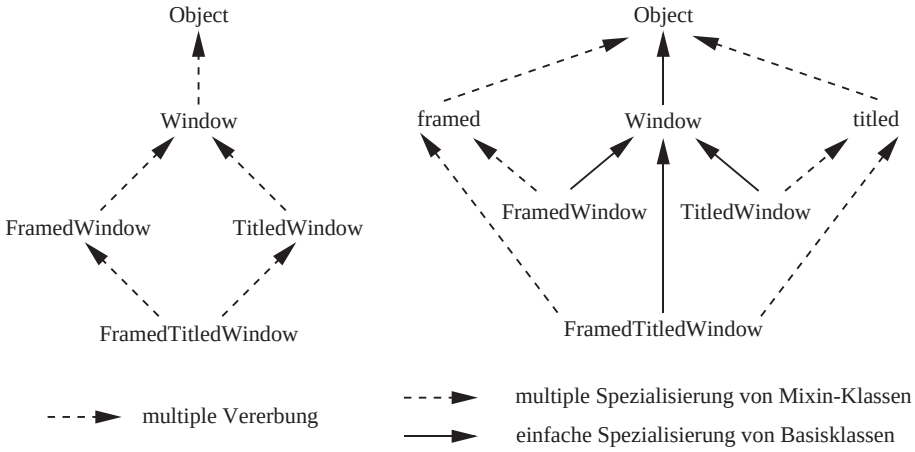


Abbildung 4.2: Multiple und Mixin-Vererbung

Die Mixin-Lösung wird oft kritisiert, weil die Klasse `FramedTitledWindow` weder Subklasse von `FramedWindow` noch von `TitledWindow` ist. Dem kann man entgegnen, daß dieser Sachverhalt uninteressant ist, wenn man konsequent generalisiert hat. Die relevanten Fragen sind dann nur, ob `FramedTitledWindow` Subklasse von `framed`, `titled` oder von `Window` ist, und zwar jeweils einzeln. Gibt es eine Stelle im Programm, wo gleichzeitig alle drei Fragen gestellt werden, ist dies ein Indikator für den Bruch der Abstraktionen.

Eine andere Frage bei Mixin-Klassen ist, ob sie überhaupt in die Klassenhierarchie unter `Object` eingehängt werden sollen. Schließlich werden sie nicht für sich allein instanziiert, sondern nur ihre Basis-Subklassen. Daher würde es reichen, wenn nur die Basisklassen von `Object` erben, nicht aber die Mixin-Klassen. Noch viel wichtiger ist aber die Frage, welche Beschränkungen für die Mixin-Klassenhierarchie gelten sollen, um die gewünschten Eigenschaften zusichern zu können. Sonst hätte man ja alle Probleme der allgemeinen multiplen Vererbung hier wieder. Am konsequentesten ist die Forderung, daß alle geerbten Mixin-Klassen disjunkt sein müssen, sowohl für erbende Mixin-Klassen wie für erbende Basisklassen. Manchmal ist diese Einschränkung jedoch zu hart, weil es unmöglich wird, für eine Reihe von Mixin-Klassen ein gewisses Mindestmaß gemeinsamen Verhaltens vorzugeben. Andererseits kann derselbe Zweck auch dadurch erreicht werden, daß man für eine Mixin-Klasse spezifiziert, welches Mindestverhalten eine Basisklasse bereitstellen muß, um vernünftigerweise von dieser Mixin-Klasse zu erben.

In Flavors gibt es dafür die Optionen `:required-class`, `:required-instance-variables` und `:required-methods`. Im Prinzip würde `:required-class` schon ausreichen. Aus Programmierersicht kann dies als eine Aussage darüber gedeutet werden, für welche (in der Regel abstrakte) Basisklasse eine Mixin-Klasse konstruiert ist. Dabei kann natürlich auch eine Subklasse der (abstrakten) Basisklasse von dieser Mixin-Klasse erben. Aus implementierungstechnischer Sicht dienten die genannten Angaben in Flavors dazu, Mixin-Klassen auch ohne sie spezialisierende Basisklassen übersetzen zu können.

Einem ähnlichen Zweck dient das sogenannte *inheritance interface* in [Flatt *et al.*, 1998]. Dort wird zwischen *atomaren* und *zusammengesetzten Mixins* unterschieden, wobei ein atomares Mixin nur ein *Interface* spezialisieren darf. In einer Interface-Definition werden Methoden spezifiziert, nicht implementiert. Erbende Mixins müssen alle vom Interface geerbten Methoden implementieren. Mögliche andere Methoden des Mixins werden nicht an seine Subklassen vererbt. Dadurch wird nicht nur das Mindestverhalten, sondern ein genaues Verhaltensprotokoll festgelegt. Zusammengesetzte Mixins entsprechen unseren Basisklassen, dürfen aber keine eigenen Feld- oder Methodenspezifikationen enthalten.

Zusammenfassend komme ich zu der Schlußfolgerung, daß Mixin-Klassen wie Basisklassen direkt oder indirekt von `Object` erben und daß man nur disjunkte Mixin-Klassen erben darf, d. h. solche, die höchstens `Object` als gemeinsame Superklasse besitzen. Diese Entscheidung impliziert auch, daß alle geerbten Felder disjunkt sind und nur explizit in einer Subklasse (Mixin- oder Basisklasse) spezialisiert werden können, im Unterschied zur allgemeinen multiplen Vererbung. Falls diese einfache Lösung nicht alle Bedürfnisse befriedigen sollte, kann das Metaobjektprotokoll verwendet werden, um sie zu erweitern oder andere Konzepte zu realisieren.

4.4 Reflektion und Spracherweiterungsprotokolle

Nachdem die Konzepte der Objektsprache ausführlich diskutiert und bestimmte Designentscheidungen getroffen wurden, sollen nun Sprachmittel zur Reflektion (in Programmen) und zur Erweiterung der Objektsprache präsentiert werden. Als Ausgangsbasis dienen die Konzepte des CLOS MOP. Unter den gesetzten Designkriterien werden jedoch andere Akzente gesetzt und abweichende Entscheidungen getroffen, die zu einer Verbesserung von Objektsystemen beitragen. Die Strategie besteht in der Vereinfachung des Metaobjektsystemkerns und der Vermeidung von Sicherheits- und Performanznachteilen für die Objektsprache sowie ihre Erweiterungen.

4.4.1 Metaobjektklassen

Die vereinfachte Hierarchie von Metaobjektklassen kann durch Einrückung wie folgt dargestellt werden:

```
Object
  MetaObject
    Class
      SimpleClass
    Field
      SimpleField
    GenericFunction
      SimpleGenericFunction
    Method
      SimpleMethod
```

Es werden also im wesentlichen vier Arten von Metaobjekten unterschieden: Klassen, Felder, generische Operationen und Methoden. Diese werden durch die abstrakten Klassen `Class`, `Field`, `GenericFunction` und `Method` repräsentiert. In ObjVLisp sind es nur Klassen, in CLOS gibt es zusätzlich Methodenkombinationsobjekte. Letztere können hier als Erweiterung definiert werden. Die Metaobjektklassenhierarchie soll möglichst so entworfen werden, daß Subklassen wirklich als Spezialisierungen ihrer Superklassen gesehen werden können. Diese Richtlinien wäre verletzt, wenn man z. B. für `Class` einfache Vererbung spezifizieren würde und in einer ihrer Subklassen multiple Vererbung. Für systemdefinierte Standard-Metaobjekte, die aus konzeptuellen und aus Effizienzgründen mit einfacher Funktionalität ausgestattet werden sollen, werden daher spezielle Metaobjektklassen mit dem Präfix `Simple` in ihren Namen definiert.

Alle Metaobjektklassen soll der Benutzer spezialisieren können. Dies steht aber im Konflikt mit gewünschten Optimierungen für Standard-Metaobjekte. Entscheidungen, ob auch sie vom Benutzer spezialisierbar sein sollen, werden erst nach der Diskussion der Implementierungstechniken getroffen. Denn erlaubt man ihre Spezialisierung, so wird es schwierig, die Einhaltung für die Optimierungen notwendiger Annahmen zu garantieren.

Das Verhalten von Metaobjekten läßt sich nicht in den einzelnen Klassen isolieren, weil immer mindestens zwei Metaobjektklassen beteiligt sind, z. B. die Klassen `Class` und `Field` am Feldzugriffsprotokoll. Die Modularisierung ihres Verhaltens erfolgt daher bezüglich

verschiedener Aspekte ihrer Funktionalität in Form von Subprotokollen. Dabei entsteht die Funktionalität in der Regel aus dem Zusammenspiel mehrerer Klassen. Es bestehen auch Wechselwirkungen zwischen den Subprotokollen, die aufeinander abgestimmt werden müssen.

4.4.2 Introspektionsprotokoll

Das Introspektionsprotokoll stellt für die unterschiedlichen Objekte entsprechende Leseoperationen bereit. Die meisten dieser Operationen werden vom Objektsystem selbst benötigt, insbesondere für die folgenden Subprotokolle. Hier stellt sich die Frage, ob die Leseoperationen als einfache oder als generische Operationen spezifiziert und realisiert werden. Eine weitere Frage ist, ob die entsprechenden Informationen durch spezifizierte Felder (dann wäre das Objektsystem voll reflektiv) oder auf andere implementierungsabhängige Weise repräsentiert werden. Bei beiden Fragen stehen die Forderungen nach Flexibilität und Uniformität vs. Sicherheit und Effizienz gegenüber. Die Speicherung interner Zustände von Metaobjekten als Feldwerte erschwert es, die Integrität des Objektsystems zu garantieren, weil Schreiboperationen dem Benutzer zugänglich werden können.

Darüber hinaus wird die Implementierung des Objektsystemkerns komplizierter, weil noch mehr zyklische Abhängigkeiten entstehen. Konzeptuell müssen dann alle Metaobjektklassen in einem Schritt in die Welt gesetzt werden, bevor ihre Protokolle funktionieren. Ähnliche Probleme ziehen auch generische Leseoperationen für Metaobjekte nach sich, weil z. B. die Ausführung einer generischen Operation schon für ihren generischen Dispatch die Ausführung anderer *generischer* Lese-Operationen erfordert. Ein potentieller Endlloszyklus muß aufgebrochen werden, was nur für systemdefinierte Metaobjekte effizient gelingt. Benutzerdefinierte Metaobjekte erleiden deutliche Effizienznachteile.

Um den Speicherbedarf zu reduzieren, sollte nur die unbedingt notwendige Information mit den Metaobjekten assoziiert werden. Dies wird um ca. 30 % verbessert, wenn das Redefinieren von Standard-Klassen nicht unterstützt zu werden braucht.

Introspektion beliebiger Objekte

Allen Objekten ist gemeinsam, daß man ihre Klassenzugehörigkeit abfragen kann. Dies geschieht mittels der Operation `classOf`. Mit jedem Objekt wird daher:

- die speziellste Klasse,

deren Instanz es ist, assoziiert. Da der systemdefinierte generische Dispatch klassenbasiert ist, muß `classOf` besonders effizient realisiert werden. Aus diesem und den oben genannten Gründen wird `classOf` auf jeden Fall als nichtgenerische Operation spezifiziert, selbst wenn alle anderen Introspektionsoperationen generisch wären.

Introspektion von Klassen und Feldern

Mit Klassenobjekten werden mindestens folgende Informationen assoziiert:

- die Klassenpräzedenzliste,

- die Liste der Felder,
- die Liste der Initialisierungsschlüsselwörter,
- die Instanzgröße.

Je nachdem, ob man das Redefinieren von Klassen oder das Debuggen von Programmen unterstützen will, werden darüberhinaus folgende Angaben benötigt:

- der Name der Klasse,
- die Liste der direkten Superklassen,
- die Liste direkt spezifizierter Felder,
- die Liste direkt spezifizierter Initialisierungsschlüsselwörter.

Auch die Instanzgröße könnte als optional betrachtet werden, weil sie prinzipiell aus der Liste der Felder ableitbar ist. Ob man sie aus Effizienzgründen als notwendig betrachtet, hängt im wesentlichen vom Entwurf des Objekterzeugungs- und Initialisierungsprotokolls ab.

Die in der Feldliste einer Klasse enthaltenen Feldobjekte müssen folgende Informationen liefern können:

- den Namen des Feldes,
- einen Ersatzwert,
- eine Ersatzwertfunktion,
- eine Leseoperation,
- eine Schreiboperation,
- ein Initialisierungsschlüsselwort.

Gegebenefalls muß ein Feld auch die Information enthalten, ob die Feldwerte konstant sind oder nicht.

Zu beachten ist, daß zwischen den Klassennamen und den Bezeichnern der Objektebene kein semantischer Zusammenhang besteht. Bezeichner spielen im Metaobjektprotokoll keine relevante Rolle, es sei denn, man würde Module zu Objekten erster Ordnung machen und entsprechende Introspektionsoperationen anbieten.

Introspektion von generischen Operationen und Methoden

Mit generischen Operationen werden mindestens folgende Informationen verknüpft:

- der Anwendungsbereich (Domain),
- die Liste der Methoden,

- die Dispatchfunktion.

Der Definitionsbereich einer generischen Operation schränkt die Anwendungsbereiche ihrer Methoden ein: sie müssen spezieller sein. Die Liste der Methoden enthält diejenigen Methoden, die zur generischen Operation gehören, also z. B. mit `define method` auf der Objektebene oder mit `addMethod` auf der Metaobjekteben hinzugefügt wurden.

Die Introspektion von Methoden muß mindestens ihren Anwendungsbereich (Domain) umfassen. Ob man einen weiteren Einblick in die innere Zusammensetzung einer Methode gewährt, bleibt noch offen. Dies wird im Zusammenhang mit Implementierungsfragen noch diskutiert.

Im Vergleich mit CLOS MOP fällt das Introspektionsprotokoll restriktiver aus. Das ermöglicht eine kompaktere Repräsentation der Metaobjekte, um den Speicherplatz effizienter zu nutzen. Darüber hinaus wird eine höhere Programmsicherheit erzielt, da keine Schreiboperationen für Metaobjekte spezifiziert werden.

4.4.3 Initialisierung von Klassen und Feldern

Die Initialisierung von Klassen und Feldern beinhaltet im wesentlichen die statisch berechenbaren Vererbungsanteile. Die Eingangsparameter der Berechnungen werden über folgende Initialisierungs-Schlüsselwörter für Klassen identifiziert:

- `name:`
- `specifiedSuperclasses:`
- `specifiedFields:`
- `specifiedKeywords:`

Die Initialisierungs-Schlüsselwörter der Felder sind:

- `name:`
- `type:`
- `initFunction:`
- `initValue:`
- `reader:`
- `writer:`
- `initKeyword:`
- `requiredInitKeyword:`

Auf der höchsten Abstraktionsstufe lassen sich acht Schritte der Klasseninitialisierung identifizieren:

1. Prüfe, ob die direkten Superklassen kompatibel mit der zu initialisierenden Klasse sind.
2. Berechne die Klassenpräzedenzliste.
3. Berechne die Initialisierungs-Schlüsselwörter, einschließlich der geerbten.
4. Berechne die Felder, einschließlich der geerbten.
5. Berechne die Instanzgröße.
6. Berechne die Lese- und Schreiboperationen für alle Felder und stelle sicher, daß auch die geerbten Lese- und Schreiboperationen auf Instanzen der zu initialisierenden Klasse korrekt funktionieren.[15]
7. Berechne ggf. die Konstruktoren.
8. Speichere die berechneten Informationen so, daß sie für die Introspektionsoperationen zugänglich sind.

Für die jeweiligen Schritte werden entsprechende generische Operationen definiert, die vom Benutzer spezialisiert werden können. Sie bilden eigenständige Subprotokolle. Die Schritte 1 bis 5 stellen das Vererbungsprotokoll im engeren Sinn dar und werden im nächsten Abschnitt behandelt. Die Schritte 6 und 7 bilden eigenständige Subprotokolle und werden in den darauf folgenden Abschnitten diskutiert. Wichtig ist, daß die spezifizierten Protokolloperationen möglichst seiteneffektfrei sind. Die Operationen der Schritte 1 bis 5 und 7 sind seiteneffektfrei. Es interessiert nur, welche Werte sie zurückliefern. Alle Größen, die sie für die Berechnung benötigen, sollten als Parameter übergeben werden. So kann auch von der Ausführungsreihenfolge abstrahiert werden. Ebenso sollte in diesen Schritten ohne Belang sein, ob einige Introspektionsoperationen auf dem zu initialisierenden Klassenobjekt bereits funktionieren.

Schritt 6 verändert möglicherweise die Feldobjekte. Bei Bedarf werden hier neue Lese- und Schreiboperationen erzeugt und mit den Feldobjekten assoziiert. Falls notwendig, werden den bereits assoziierten Lese- und Schreiboperationen weitere Methoden hinzugefügt. Dies ist z. B. dann erforderlich, wenn die geerbten und möglicherweise optimierten Methoden nicht für Instanzen der zu initialisierenden Klasse verwendet werden dürfen.

Schritt 8 löst definitiv einen Seiteneffekt aus: er verändert den Zustand des zu initialisierenden Klassenobjekts. Ein Problem, das sich hier wie in Schritt 6 stellt, ist, daß man aus Sicherheitsgründen keine Schreiboperationen für Metaobjekte offenlegen möchte. Daraus folgt aber, daß die systemdefinierte Initialisierungsmethode für Klassen auf jeden Fall als *ganzes* ausgeführt werden muß, um Schritt 8 zu erledigen. In benutzerdefinierten Initialisierungsmethoden muß also unbedingt ein `callNextMethod` ausgeführt werden, weil sonst die Integrität von Metaobjekten gefährdet wäre.[16] Ähnliches gilt auch für die Operation `computeAndEnsureFieldAccessors`, die Schritt 6 umfaßt.

[15] Bei generischen Lese- und Schreiboperationen müssen ggf. neue Methoden hinzugefügt werden.

[16] Wohl aus einer ähnlichen Überlegung werden in Java die dort sogenannten Konstruktoren aller Superklassen automatisch ausgeführt. Ich würde sie lieber als Initialisierungsmethoden bezeichnen. In CLOS MOP sollen vom Benutzer entsprechend nur After-Methoden definiert werden.

```
compatibleSuperclasses? cl directSuperclasses → boolean
  compatibleSuperclass? cl superclass → boolean
computeCPL cl directSuperclasses → (cl*)
computeInheritedKeywords cl directSuperclasses → ((keyword*)*)
computeKeywords cl directKeywords inheritedKeywords → (keyword*)
computeInheritedFields cl directSuperclasses → ((field*)*)
computeFields cl fieldSpecs inheritedFields → (field*)
  entweder
  computeNewField cl fieldSpec → field
    computeNewFieldClass cl fieldSpec → fieldClass
  oder
  computeSpecializedField cl inheritedFields fieldSpec → field
    computeSpecializedFieldClass cl inheritedFields fieldSpec → fieldClass
computeInstanceSize cl fields → integer
computeAndEnsureFieldAccessors cl fields inheritedFields → (field*)
  ...
```

Abbildung 4.3: Aufrufstruktur der Klasseninitialisierung

Abbildung 4.3 zeigt die logische Aufrufstruktur von generischen Operationen während der Initialisierung von Klassenobjekten. Durch Einrückung wird dabei angezeigt, wenn eine Operation von der darüberstehenden aufgerufen wird. Die Schritte 1 bis 8 werden nun genauer erläutert.

4.4.4 Vererbungsprotokoll

Auf der Objektebene von Programmiersprachen wurden verschiedene Vererbungskonzepte bereits ausführlich diskutiert. Ein Ergebnis dieser Diskussion war, dass es noch keine endgültige und alle befriedigende Antwort auf die Frage gibt, welches das beste Vererbungskonzept sei. Einen Ausweg aus diesem Dilemma bietet das Vererbungsprotokoll auf der Metaobjektebene. Dieses erlaubt und unterstützt nicht nur *ein*, sondern *mehrere*, auch benutzerdefinierte Vererbungskonzepte. Dabei existieren diese nicht nur nebeneinander, sondern sie stehen sozusagen unter einem gemeinsamen Dach. Ein guter Entwurf des Vererbungsprotokolls sollte auf verschiedenen Granularitätsstufen eine einfache Spezialisierung bereits definierter Teilkonzepte unterstützen.

Zulässigkeit direkter Superklassen

Für die Überprüfung der Kompatibilität der direkten Superklassen (Schritt 1) werden zwei generische Operationen in zwei Stufen definiert: `compatibleSuperclasses?` auf der oberen Ebene, um auch multiple bzw. Mixin-Vererbung zu erfassen. Sie ruft seinerseits `compatibleSuperclass?` auf der nächstniedrigeren Ebene auf, um die paarweise Kompatibilität zu prüfen. Der generische Dispatch erfolgt bei `compatibleSuperclasses?` allein anhand des ersten Parameters (einfacher Dispatch). Die Klasse des zu initialisierenden Klassenobjekts bestimmt also die Methodenauswahl. Bei der paarweisen Prüfung in `compatibleSuperclass?` bestimmen beide Parameter die Methodenauswahl. Dies ist ein

typisches Beispiel für multiplen Dispatch. Dadurch wird es möglich, orthogonale Erweiterungen modular zu organisieren, z. B. dann, wenn weitere Arten von Superklassen für eine Klassenart als kompatibel erklärt werden sollen.

Berechnung der Klassenpräzedenzliste

Die Berechnung der Klassenpräzedenzliste (Schritt 2) in der generischen Operation `computeCPL` stellt die Verbindung zur Methodenauswahl und dem generischen Dispatch her. Dort ist dann nur noch die Klassenpräzedenzliste maßgeblich. Die direkten Superklassenbeziehungen müssen nicht mehr betrachtet werden. Der generische Dispatch erfolgt anhand des ersten Parameters, also der zu initialisierenden Klasse. Die spezifizierten direkten Superklassen werden als zweiter Parameter übergeben. Die Berechnung der Klassenpräzedenzliste kann daher ohne die Annahme einer bereits durchgeführten Teilinitialisierung des neuen Klassenobjekts erfolgen. In CLOS MOP werden die direkten Superklassen nicht als Argument durchgereicht, so daß in benutzerdefinierten `compute-class-precedence-list`-Methoden die entsprechende Introspektionsoperation `class-direkt-superklasses` aufgerufen werden muß. Schon durch diese Designentscheidung muß in CLOS mehr Speicherplatz für die Klassenrepräsentation vorgesehen werden. Ebenso hat man die sonst unnötige Einschränkung bewirkt, daß die Berechnung der Klassenpräzedenzliste nicht vor der Teilinitialisierung passieren darf.

Berechnung der Initialisierungs-Schlüsselwörter

Die Berechnung der Initialisierungs-Schlüsselwörter (Schritt 3) erfolgt in den generischen Operationen `computeInheritedKeywords` und `computeKeywords`. `computeInherited-Keywords` berechnet die zu erbenden Schlüsselwörter, `computeKeywords` berechnet ausgehend von den geerbten und den neu spezifizierten die maßgebliche (engl. *effective*) Liste aller Initialisierungs-Schlüsselwörter für die zu initialisierende Klasse. Beide Operationen diskriminieren über den ersten Parameter: das zu initialisierende Klassenobjekt. Der zweite Parameter von `computeInheritedKeywords` ist die Liste der direkten Superklassen, wie bei `computeCPL`. Der zweite Parameter von `computeKeywords` ist die Liste der neu spezifizierten Initialisierungs-Schlüsselwörter. Über den dritten Parameter wird das Ergebnis von `computeInheritedKeywords` übergeben. So kann in benutzerdefinierten Spezialisierungen z. B. eine Filterung der zu erbenden Initialisierungs-Schlüsselwörter einfach realisiert werden: Es muß nur eine speziellere `computeInheritedKeywords`-Methode definiert werden.

Berechnung der Felder

Für die Berechnung der Felder (Schritt 4) stehen in drei Stufen die generischen Operationen `computeInheritedFields`, `computeFields`, `computeNewField`, `computeNewFieldClass`, `computeSpecializedField` und `computeSpecializedFieldClass` zur Verfügung. Analog zu `computeInheritedKeywords` bei den Initialisierungs-Schlüsselwörtern berechnet `computeInheritedFields` die zu erbenden Felder, ausgehend von den gleichen Parametern. Analog zu `computeKeywords` berechnet `computeFields` die maßgebliche Feldliste für die zu initialisierende Klasse. Ihr erster Parameter ist diese Klasse selbst, der zweite ist die Liste der neuen Feldspezifikationen und der dritte sind die zu erbenden Felder, welche

von `computeInheritedFields` berechnet wurden. `computeFields` unterscheidet zwischen neuen und spezialisierten Feldern und ruft für jedes Feld jeweils die generische Operation `computeNewField` bzw. `computeSpecializedField` auf. Felder werden eindeutig über ihre Namen identifiziert. Die Parameter von `computeNewField` sind das Klassenobjekt, über das diskriminiert wird, und die Feldspezifikation als Liste mit Schlüsselwörtern und korrespondierenden Werten. `computeSpecializedField` erhält zusätzlich als Parameter die Liste der zu erbenden Felder, die im Fall der einfachen und der oben vorgeschlagenen Mixin-Vererbung aus einem Element besteht. Das Protokoll soll aber auch die allgemeine Verbung wie in CLOS ermöglichen bzw. in Erweiterungen unterstützen können. Um festzulegen, von welcher Feldklasse ein Feldobjekt erzeugt werden soll, wird die generische Operation `computeNewFieldClass` bzw. `computeSpecializedFieldClass` aufgerufen. Die systemdefinierten Methoden liefern als Ergebnis die Klasse `SimpleField` zurück. Auf diese Weise kann der Benutzer sehr einfach und gezielt das Protokoll spezialisieren.

Im nächsten Abschnitt wird Schritt 6, im übernächsten Abschnitt Schritt 7 diskutiert. Zuletzt wird die Rolle von Schritt 5 für das Allokationsprotokoll behandelt. Schritt 8 wird in der Spezifikation nicht offengelegt, das Schreiben von Metaobjektinformationen ist ein Implementierungsdetail.

4.4.5 Feldzugriffsprotokoll

Im Unterschied zu CLOS MOP, dessen Feldzugriffsprotokoll so spezifiziert ist, daß zeitaufwendige Berechnungen zur Laufzeit erfolgen müssen, wird hier ein Konzept entwickelt, daß solche Berechnungen in die Klasseninitialisierungsphase verlagert. Diese erfolgt in der Regel zur Programmladezeit. Die überflüssige Erzeugung mehrerer Lese- und Schreiboperationen für dieselben Felder wird vermieden. Intern wird für jedes Feld genau eine Lese- und eine Schreiboperation berechnet. Will man das Feldzugriffsverhalten spezialisieren, so müssen speziellere Methoden definiert werden, welche entsprechende Zugriffsoperationen berechnen. Dies ist ein grundleger Unterschied zu CLOS MOP, wo die Zugriffsoperationen selbst generisch sind und spezialisiert werden können.

In meinem Entwurf müssen die Zugriffsoperationen nicht unbedingt generisch sein. Insbesondere die einfache Vererbung im Objektsystemkern erlaubt es, mit nichtgenerischen Lese- und Schreiboperationen auszukommen. Generisch und somit spezialisierbar sind aber die Operationen, die die Zugriffsoperationen berechnen. Diese Lösung setzt allerdings voraus, daß Funktionen bzw. Prozeduren Objekte erster Ordnung sind und daß es möglichst auch sogenannte *Closures* gibt. Letztere sind funktionale Objekte, die ihre lexikalische Umgebung zum Zeitpunkt ihrer Erzeugung einschließen. Man kann sie auch als funktionale Objekte mit eigenem Gedächtnis bzw. Zustand ansehen.

Die das Feldzugriffsprotokoll umfassende generische Operation ist `computeAndEnsureFieldAccessors`. Als Parameter werden ihr die zu initialisierende Klasse, die für diese Klasse bereits berechneten Felder sowie die geerbten Felder übergeben. Diskriminiert wird über den ersten Parameter, also über die Klasse des Klassenobjekts. Abbildung 4.4 zeigt den Ausschnitt der Klasseninitialisierung, der das Feldzugriffsprotokoll ausmacht.

Dieses Protokoll ist komplexer, als es für einfache Vererbung notwendig wäre. Sein Vorteil ist die einfache Erweiterung und Spezialisierung auf verschiedenen Granularitätsstufen durch den Benutzer. Insbesondere die Realisierung der multiplen bzw. der Mixin-

```
computeAndEnsureFieldAccessors cl fields inheritedFields → (field*)
  computeFieldReader cl field fields → function
  computeFieldWriter cl field fields → function
  ensureFieldReader cl field fields reader → function
  ensureFieldWriter cl field fields writer → function
```

Abbildung 4.4: Berechnung der Feldzugriffsoperationen

Vererbung ist sehr einfach möglich. Zunächst geht es hier natürlich nicht um die Frage der einfachen vs. multiplen Vererbung, sondern mehr um die Frage, ob die Zugriffsoperationen generisch sein sollen oder nicht. Davon hängt aus Benutzersicht ab, ob sie auf der Objektebene, und aus Sprachimplementierersicht, ob sie auf der Metaobjektebene spezialisierbar sind. Allerdings ergeben sich aus der Entscheidung über das Vererbungskonzept entsprechende Pro- und Kontra-Argumente, wie die Zugriffsoperationen berechnet werden.

Da Zugriffsoperationen vererbt werden, muß sichergestellt werden, daß sie auch auf Instanzen der Subklassen korrekt und effizient funktionieren. Daher teilen sich die Berechnungsoperationen des Feldzugriffsprotokolls in zwei Arten auf: solche, die neue Lese- und Schreiboperationen berechnen (`computeFieldReader` und `computeFieldWriter`), und solche, die sicherstellen, daß sie auf Instanzen der zu initialisierenden Klasse anwendbar werden (`ensureFieldReader` und `ensureFieldWriter`).

`computeFieldReader` und `computeFieldWriter` erhalten als Parameter die zu initialisierende Klasse, das Feld, für das die Zugriffsoperation berechnet werden soll, sowie die Liste aller Felder. Der letzte Parameter ist wichtig, weil bei der Berechnung der Zugriffsoperationen nicht nur die lokalen Eigenschaften eines Felds selbst, sondern auch alle Felder gemeinsam betreffende Eigenschaften, relevant sind. Beispielsweise ergeben sich die Positionen von Feldwerten in Instanzen aus den Positionen der Felder in der Feldliste. Jedenfalls solange alle Felder instanzspezifisch alloziert werden, also z. B. wenn es keine *shared slots* gibt.

Die ersten drei Parameter von `ensureFieldReader` und `ensureFieldWriter` sind die gleichen wie eben. Sie erhalten darüberhinaus einen weiteren Parameter, nämlich die schon berechnete Zugriffsoperation. Wenn diese nichtgenerisch ist, dann kann sie auch nicht weiter angepaßt werden. Es kann nur geprüft werden, ob sie paßt, und falls nicht, muß ein Fehler gemeldet werden. Wenn es eine generische Operation ist, kann es sein, daß noch keine Methoden hinzugefügt wurden oder daß die für Instanzen der Superklassen hinzugefügten Methoden auf Instanzen der zu initialisierenden Klasse nicht anwendbar sind. In beiden Fällen muß eine neue Methode hinzugefügt werden.

Im Falle einfacher Vererbung können auch nichtgenerische, optimalerweise mit einem festen Index zugreifende Lese- und Schreiboperationen so realisiert werden, daß sie immer auch für Instanzen der Subklassen gültig bleiben. Die `ensureFieldReader`- und `ensureFieldWriter`-Methoden für Klassenobjekte der Klasse `SimpleClass` brauchen daher nichts zu tun. Die nächsten Granularitätsebenen spielen sich daher innerhalb der generischen Operationen `computeFieldReader` und `computeFieldWriter` ab. Dies zeigt Abbildung 4.5.

Mit den generischen Operationen `computePrimitiveReaderUsingField`, `computePrimi-`

```
computeAndEnsureFieldAccessors cl fields inheritedFields → (field*)
  computeFieldReader cl field fields → function
    computePrimitiveReaderUsingField field cl fields → function
      computePrimitiveReaderUsingClass cl field fields → function
  computeFieldWriter cl field fields → function
    computePrimitiveWriterUsingField field cl fields → function
      computePrimitiveWriterUsingClass cl field fields → function
  ensureFieldReader cl field fields reader → function
  ensureFieldWriter cl field fields writer → function
```

Abbildung 4.5: Berechnung der Feldzugriffsoperationen bei einfacher Vererbung

`tiveWriterUsingField`, `computePrimitiveReaderUsingClass` und `computePrimitiveWriterUsingClass` will man gezielt Aspekte eines Felds bzw. Aspekte der zu initialisierenden Klasse in die Berechnung der Lese- und Schreiboperationen einfließen lassen. Im Standardfall, d. h. wenn das Klassenobjekt Instanz von `SimpleClass` ist und das Feldobjekt Instanz von `SimpleField` ist, gibt `computePrimitiveReaderUsingField` das Ergebnis von `computePrimitiveReaderUsingClass` einfach weiter. Analog verhalten sich im Standardfall `computePrimitiveWriterUsingField` und `computePrimitiveWriterUsingClass`.

Beispielsweise kann `computePrimitiveWriterUsingField` die Feldannotation `type:` in die Berechnung der Schreiboperation einbringen, während `computePrimitiveReaderUsingClass` die primitive Allokationsart und die Feldposition bzw. den primitiven Zugriff (z. B. über einen Index) realisiert. Hier befindet sich die Schnittstelle des Feldzugriffsprotokolls zum Allokationsprotokoll 4.4.6.

Im Falle allgemeiner multipler Vererbung und der Mixin-Vererbung kann sich die Position eines Feldes von Klasse zu Subklasse ändern. Daher bietet es sich an, die Zugriffsoperationen als generische Operationen zu realisieren. Dabei werden für die Klassen mit verschiedenen Positionen desselben Feldes jeweils unterschiedliche Methoden definiert, die den Zugriff jeweils mit einem festen Index durchführen. Der ohnehin vorhandene Mechanismus des generischen Dispatches wird ausgenutzt, um für eine aktuelle Instanz die richtige Methode für den Feldzugriff auszuwählen. Je seltener sich die Feldpositionen ändern, um so weniger Methoden müssen für eine Zugriffsoperation definiert werden und um so besser kann der generische Dispatch optimiert werden. Hier greife ich der Diskussion der Implementierungstechniken im nächsten Kapitel vor, weil die Konzepte des Metaobjektprotokolls eng mit Implementierungs- und Optimierungsfragen zusammenhängen.

Entscheidet man sich nun für generische Lese- und Schreiboperationen, so verschiebt sich der Berechnungsaufwand von der Erzeugung generischer Zugriffsoperationen in `computeFieldReader` und `computeFieldWriter` hin zur Sicherstellung ihrer Anwendbarkeit in den Protokoll-Opertionen `ensureFieldReader` und `ensureFieldWriter`. Dies verdeutlicht Abbildung 4.6.

Eine angemessenere Aufgabenverteilung erhält man, wenn `computeFieldReader` und `computeFieldWriter` leere generische Zugriffsoperationen (ohne assoziierte Methoden) erzeugen und als Ergebnis zurückliefern. Zugriffsmethoden werden erst in den Protokoll-Operationen `ensureFieldReader` und `ensureFieldWriter` bei Bedarf erzeugt und den

```
computeAndEnsureFieldAccessors cl fields inheritedFields → (field*)
  computeFieldReader cl field fields → function
  computeFieldWriter cl field fields → function
  ensureFieldReader cl field fields reader → function
    computePrimitiveReaderUsingField field cl fields → function
      computePrimitiveReaderUsingClass cl field fields → function
  ensureFieldWriter cl field fields writer → function
    computePrimitiveWriterUsingField field cl fields → function
      computePrimitiveWriterUsingClass cl field fields → function
```

Abbildung 4.6: Berechnung der Feldzugriffsoperationen bei multipler Vererbung

vorher berechneten Zugriffsoperationen hinzugefügt. Daher werden auch erst hier ggf. die Protokoll-Operationen `computePrimitiveReaderUsingField` und `computePrimitiveReaderUsingClass` bzw. `computePrimitiveWriterUsingField` und `computePrimitiveWriterUsingClass` aufgerufen. Falls es sich um geerbte Felder handelt, sind die bereits vorhandenen Methoden möglicherweise auch für Instanzen der zu initialisierenden Klasse gültig. Dann brauchen keine neuen Methoden berechnet und hinzugefügt werden.

Zusammenfassend kann man sagen, daß es natürlich schwierig ist, ein Feldzugriffsprotokoll zu entwerfen, das einen breiten Design- und Erweiterungs-Raum abdeckt: von einfacher bis hin zur Mixin-Vererbung, von nichtgenerischen Zugriffsoperationen der Objektebene bis hin zu generischen Zugriffsoperationen, die auch vom Benutzer der Objektebene erweiterbar sind, wie in CLOS MOP. Das hier entwickelte Protokoll ist natürlich nicht als ein letztes Wort zu diesem Thema zu verstehen. Es ist eine Kombination aus einer einfachen und effizienten Lösung auf der einen Seite sowie einer offenen Lösung, die prinzipiell beliebige Konzepte als Erweiterungen zuläßt, auf der anderen Seite. Dabei kann man aber nicht gleichermaßen alle beliebigen Wünsche potentieller Nutzer unterstützen. Auch die Offenheit muß seine Grenzen haben, um noch Zuverlässigkeitsgarantien für den Objektsystemkern und seine Erweiterungen sowie für deren Anwendungen geben zu können. Dieser Aspekt wird im Abschnitt 4.5 aufgegriffen.

4.4.6 Allokations- und Initialisierungsprotokoll

Die höheren Operationen des Allokations- und Initialisierungsprotokolls sind

- `allocate`, die auf der Objektebene als nichtgenerische und somit nichterweiterbare Operation spezifiziert wurde,
- `initialize`, die schon auf der Objektebene als generische und somit erweiterbare Operation spezifiziert wurde, und
- `make` als Zusammenfassung der beiden ersten Operationen.

Hier gibt es wie beim Feldzugriffsprotokoll zwei alternative Konzepte. In der ersten Alternative diskriminieren `allocate` und `initialize`, wie in CLOS, zur Instanzerzeugungszeit. Dies entspricht der Diskriminierung in `slot-value-using-class` im Feldzugriffsprotokoll von CLOS MOP. Die Vereinfachung gegenüber CLOS ist, daß die Operation `make`

```
make cl initarg* → object
allocate cl initargs → object
initialize object initargs → object
computeInstanceSize cl fields → integer
classInstanceSize cl → integer
computeConstructor cl initkeywords → function
computeAllocator cl initkeywords → function
computeInheritedInitializer cl directSuperclasses → list
computeInitializer cl directInitMethod inheritedInitMethods → list
```

Abbildung 4.7: Abstrakte Allokationsoperationen

```
primitiveAllocate cl → primitiveAllocatedObject
primitiveClassOf primitiveAllocatedObject → class
setPrimitiveClassOf primitiveAllocatedObject cl
primitiveRef primitiveAllocatedObject index → value
setPrimitiveRef primitiveAllocatedObject index newValue
```

Abbildung 4.8: Primitive Allokationsoperationen

nicht generisch ist. Sie faßt nur beide generische Operationen `allocate` und `initialize` zusammen. Auf diese Weise ist das generische Objekterzeugungsverhalten modular und orthogonal. Es gibt im Unterschied zu CLOS genau eine Operation, um den gewünschten Verhaltensaspekt spezialisieren zu können.

Die zweite Alternative greift die Idee aus dem Feldzugriffsprotokoll auf. Dabei werden die generischen Operationen `computeAllocator`, `computeInheritedInitializer`, `computeInitializer` und `computeConstructor` bereitgestellt, die zur Klasseninitialisierungszeit ausgeführt werden. Alle drei Operationen `allocate`, `initialize` und `make` sind dann nichtgenerisch. Ihre Aufgabe ist nur, die mit der Klasse assoziierten und schon vorher berechneten Operationen auszuführen. Auf der Objektebene muß eine Klassenannotation `initialize:` eingeführt werden, um benutzerdefinierte Initialisierungsmethoden trotzdem zu unterstützen. Vorteil dieser Lösung ist die weitere Effizienzverbesserung des Laufzeitverhaltens. Sie wird in den nächsten Kapiteln weiter diskutiert.

Abbildung 4.7 zeigt die abstrakteren Operationen im Überblick.

Nun werden zunächst die primitiveren Operationen des Allokationsprotokolls vorgestellt. Die folgenden Allokationsprimitive werden benötigt, um andere als die systemdefinierte Art zu unterstützen, wie man Objekte im Speicher repräsentiert. Beispiele von Systemerweiterungen, die dieses benötigen, sind persistente Objekte, reklassifizierbare Objekte oder auch redefinierbare Klassen mit automatischer Anpassung bereits erzeugter Instanzen. Dabei müssen Sicherheitsaspekte besonders berücksichtigt werden.

Im Prinzip geht es darum, typisierte Vektoren zur Verfügung zu stellen, deren Typ gelesen, gesetzt und verändert werden kann. Natürlich wird auch das Lesen und Schreiben der Felder mittels eines Index benötigt. Abbildung 4.8 zeigt die entsprechenden Operationen.

Um eine unzulässige Introspektion oder gar Destruktion von Instanzen systemdefinierter sowie benutzerdefinierter Klassen zu unterbinden, muß der Objektsystemkern die Verwendung der Allokationsprimitive sorgfältig kontrollieren. Dies wird dadurch erreicht, daß die mittels `primitiveAllocate` erzeugten Objekte von allen anderen unterscheidbar sein müssen. Die primitiven Lese- und Schreiboperationen dürfen nur auf mittels `primitiveAllocate` erzeugten Objekte angewandt werden. Anderenfalls meldet das System einen Fehler. Die Größe, d. h. die Anzahl der Vektorfelder, primitiv allozierter Objekte ist implizit durch das Klassenobjekt, den einzigen Parameter von `primitiveAllocate`, festgelegt. Alle direkten Instanzen einer Klasse haben aus Systemsicht dieselbe Größe. Sie wird mittels `computeInstanceSize` berechnet und mittels `classInstanceSize` abgefragt. Da es keine entsprechende Schreiboperation gibt, liefert `classInstanceSize` für eine Klasse immer denselben einmal berechneten Wert. Objekte mit einer anderen Größe, als der von der Klasse vorgegebenen, können erst gar nicht entstehen. Wird die Klasse eines primitiv allozierten Objekts mit `setPrimitiveClassOf` verändert, so wird geprüft, ob `classInstanceSize` für die neu zu setzende Klasse den gleichen Wert[17] liefert wie die alte Klasse des primitiv allozierten Objekts. Nur bei positivem Testausgang wird die Änderung vorgenommen. Auf diese Weise kann auch das Reklassifizieren von Objekten unter Einhaltung minimaler Anforderungen an die Typsicherheit realisiert werden. Mögliche Typinferenzen des Compilers können durch das Metaobjektprotokoll, insbesondere die primitive Allokation von Objekten, nicht konterkariert werden.

4.4.7 Initialisierung generischer Operationen und Methoden

Die Sprachmittel der Objektebene lassen nur eine statische Definition von Klassen, generischen Operationen und Methoden zu. Das heißt beispielsweise, daß man durch statische Analyse schon zur Übersetzungszeit alle Methoden einer generischen Operation kennt. Bedenkt man jedoch, daß es wünschenswert ist Software-Komponenten auch im Objektcode-Format zu spezialisieren, so macht es Sinn, auf der Metaobjektebene von einer inkrementellen Erweiterung generischer Operationen um neue Methoden auszugehen. Eine Möglichkeit, dem Rechnung zu tragen, ist, die Initialisierung von generischen Operationen bei jeder Erweiterung erneut durchzuführen. Auf der anderen Seite sollten überflüssige bzw. sich wiederholende Berechnungen vermieden werden.

Darüber hinaus stellt sich die prinzipielle Frage, ob Programmcode zur Laufzeit generiert, übersetzt und dynamisch hinzugebunden werden soll. Dies kann zu gravierenden Performanznachteilen führen. In meinem Sprachdesign soll daher keine Entscheidung letzteres erzwingen. Darin liegt ein wichtiger Unterschied zu CLOS. Allerdings muß das Kind nicht mit dem Bade ausgeschüttet werden.

In der Tradition funktionaler Sprachen weiß man die Grenze scharf zu ziehen. Funktionale Objekte können sehr wohl dynamisch, also zur Laufzeit, erzeugt werden, ohne daß es dabei notwendig ist, Quellcode zu übersetzen. Dafür muß das Funktionsmuster zur Übersetzungszeit feststehen. Zur Laufzeit können also funktionale Objekte erzeugt werden, deren Berechnungsvorschrift identisch, aber deren eingeschlossene Umgebung unterschiedlich ist. Dies kann z. B. ausgenutzt werden, um Lese- und Schreiboperationen auf der

[17] Geht es nur darum, Zugriffe mit unzulässigen Indexwerten auszuschließen, könnte die Bedingung auf >= gelockert werden.

Metaobjekteben dynamisch zu berechnen, wobei die Position eines Felds eingeschlossen wird und somit nicht als Parameter zum Zeitpunkt des Aufrufs der berechneten Operation übergeben werden muß. Die entsprechenden Sprachmittel auf der Metaobjektebene sind analog zur syntaktischen Spezialform `lambda` aus Scheme:

- `method lambda`, um anonyme Methoden zu erzeugen, und
- `generic lambda`, um anonyme generische Operationen zu erzeugen.

Als systemdefinierte Initialisierungs-Schlüsselwörter sind für Methoden

- `domain:` und
- `methodClass:`

sowie für generische Operationen

- `domain:`,
- `methodClass:` und
- `methods:`

zu nennen. Weitere benutzerdefinierte Initialisierungs-Schlüsselwörter werden an die Initiallisierungsmethoden weitergereicht und können dort benutzerspezifisch behandelt werden.

Falls die Programmiersprache Makros zur Verfügung stellt, können die Spezialformen `method lambda` und `generic lambda` prinzipiell auf die einfachere Form `lambda` zurückgeführt werden. Weitere Aspekte hierzu werden im nächsten Kapitel behandelt.

4.4.8 Methodenauswahl und generischer Dispatch

Die Methodenauswahl und der generische Dispatch, wie sie vom Objektsystem-Kern bereitgestellt werden, sind klassenbezogen. Das heißt, daß Methoden klassenspezifisch definiert werden. Bei einer Operationsanwendung (oder Aufruf) bestimmen die Argumentklassen, welche der mit der generischen Operation assoziierten Methoden anwendbar sind, wie sie gemäß ihrer Spezifizität geordnet werden sollen und welche Methode (zuerst) ausgeführt wird. Methodenkombination sowie instanzspezifische Methoden (eql-Methoden) wie in CLOS werden vom Objektsystem-Kern nicht direkt untertützt. Diese Konzepte können jedoch unter Ausnutzung des hier beschriebenen Protokolls als Erweiterung realisiert werden, siehe Seite 265. Abbildung 4.9 zeigt die entsprechenden Operationen im Überblick.

Die generische Operation `addMethod` fügt dem ersten Argument, einer generischen Operation, das zweite Argument, eine Methode, hinzu. Die Spezifikation dieser Operation erfordert besondere Vorsicht. Soll es z. B. erlaubt sein, eine neue Methode hinzuzufügen, wenn bereits eine mit demselben Bereich hinzugefügt wurde? Falls ja, wie kann man dann noch die Integrität des Objektsystemkerns garantieren? Die gesetzten Kriterien der Robustheit und Effizienz sprechen eindeutig dagegen. Im Unterschied zu CLOS muß der

`addMethod` *genericOp method* → *genericOp*
`computeMethodLookupFunction` *genericOp domain* → *function*
`computeDiscriminatingFunction` *genericOp domain lookupFn methods* → *function*
`callMethod` *method nextMethods arg** → *result*
`applyMethod` *method nextMethods arg** *args* → *result*

Abbildung 4.9: Operationen auf generischen Operationen und Methoden

Bereich der hinzuzufügende Methode spezieller oder gleich dem der generischen Operation sein; eine Methode mit demselben Bereich darf noch nicht hinzugefügt worden sein[18]. Die Klasse der hinzuzufügende Methode muß spezieller oder gleich der entsprechenden Spezifikation `methodClass:` der generischen Operation sein. Das Hinzufügen einer neuen Methode verlangt, daß bei einer anschließenden Anwendung der generischen Operation auf entsprechende Argumente die neue Methode auch zur Anwendung kommt.

Je nach Optimierungsansatz muß die Dispatch-Funktion dafür neu berechnet werden. Dies geschieht durch Aufruf der generische Operation `computeDiscriminatingFunction`. Ihr Resultat wird bei jedem Aufruf der erweiterten generischen Operation zuerst ausgeführt.

`computeMethodLookupFunction` berechnet für eine gegebene generische Operation (erstes Argument) sowie einen gegebenen Bereich (zweites Argument) eine nicht-generische Lookup-Funktion, die mit der generischen Funktion bzw. mit der Dispatch-Funktion assoziiert wird. Angewandt auf gegebene Argumente eines generischen Funktionsaufrufs liefert sie eine geordnete Liste der darauf anwendbaren Methoden. Sowohl `computeMethodLookupFunction` als auch `computeDiscriminatingFunction` können vom Benutzer spezialisiert werden. Um neue Dispatch-Konzepte portabel unter Ausnutzung des Metaobjektprotokolls implementieren zu können, werden die speziellen Operationen bzw. Spezialformen `callMethod` sowie `applyMethod` zur Verfügung gestellt. Die erste entspricht der normalen Operationsanwendung, während die zweite der Scheme-Funktion `apply` entspricht. Diese läßt im Unterschied zu CommonLisp Quellcode in Listenform als Argumente nicht zu. Die Kritierien der Effizienz und der Robustheit verlangen diese Einschränkung. Man benötigt `callMethod` und `applyMethod` (über das einfache `apply` hinaus), um die Realisierung des `callNextMethod` Konzepts zu verstecken. Dies wird im nächsten Kapitel noch ausführlich behandelt.

4.5 Abschlußeigenschaften der Sprachkonstrukte

Es mag einige Leser verwundern, daß in einer objektorientierten Programmiersprache mit einem Metaobjektprotokoll gleichzeitig nach Abschlußeigenschaften der Sprachkonstrukte gefragt wird. Hier kommt erneut meine Grundhypothese zum Ausdruck, daß man bei anwendungsorientierten universellen Programmiersprachen nicht ein Ziel unter Vernachlässigung aller anderen anstreben sollte. Vielmehr kommt es auf die Verhältnismäßigkeit der Mittel an. Der Programmierer muß die Wahl haben, welchen Grad an Offenheit seine

[18] Diese Beschränkung muß natürlich nicht zur Programmentwicklungszeit für entsprechende Entwicklungswerkzeuge gelten. Hier geht es jedoch um die Semantik der Programmiersprache, die klar vom Verhalten der Programmierumgebung zu trennen ist.

Software-Bausteine haben sollen. Der Compilerbauer braucht eine klare Sprachsemantik, um über Zulässigkeit und Grad von Optimierungen einfach entscheiden zu können. Vor der ausführlichen Behandlung von Implementierungstechniken im nächsten Kapitel sollen die Abschlußeigenschaften der Sprachkonstrukte aus Sicht des Anwendungsprogrammierers diskutiert werden.

Neben der besseren Performanz sichern wohldefinierte Abschlußeigenschaften ein wesentlich höheres Maß an Robustheit von Software-Systemen. Auch Aspekte des *Information-Hiding* finden hier Eingang. Spätestens mit den Konzepten der strukturierten Programmierung wurden angemessene Abschlüsse von Programmteilen angestrebt. Herausgebildet haben sich die Konzepte der *Sichtbarkeit und Lebensdauer von Bindungen.*

4.5.1 Abschlußeigenschaften von Bindungen

Man unterscheidet die lexikalische und die dynamische Bindungsart. Wurde früher in der Lisp-Gemeinde noch darüber gestritten, welche die richtige sei, so ist es heute keine Frage, daß man als Regelfall lexikalische Bindungen bevorzugt, während dynamische Bindungen eher die Ausnahme bilden. Weiterhin unterscheidet man zwischen lokalen und globalen Bindungen. Lokale Bindungen haben in der Regel eine kürzere Lebensdauer. Sie werden für die Dauer der Ausführung des nächstumschließenden Blocks angelegt und anschließend wieder aufgelöst. Blöcke regeln die Sichtbarkeit von Konstanten- und Variablenbindungen. Sie haben einen lexikalischen Anfang und ein Ende und schließen somit Bindungen ab. Entsprechend werden Prozeduren, Funktionen bzw. Methoden als implizite Blöcke aufgefaßt. Die nächstgrößeren Einheiten bilden Module. In einigen Sprachen, wie Java, auch Klassen, dann aber in der Rolle als Module.

In strikt modularen Sprachen gibt es außer Modulbindungen keine globalen Bindungen. Es sind die Modulbindungen, die zwischen Modulen importiert und exportiert werden sollten. Die Lebensdauer von Modulbindungen bzw. globalen Bindungen erstreckt sich über die gesamte Programmausführung. Die Lebensdauer von lokalen Bindungen ist an die Ausführungsdauer des entsprechenden Blocks gebunden. Eine Besonderheit bilden sogenannte *Closures*. Dies sind funktionale Objekte, die zum Zeitpunkt ihrer Erzeugung lokale oder Modulbindungen einschließen und über die Aktivationsdauer des lexikalisch umschließenden Blocks hinweg aufrechterhalten. Dabei können auch mehrere funktionale Objekte dieselben Bindungen einschließen und somit gemeinsam nutzen[19].

Eine wichtige Eigenschaft von Bindungen ist die, ob sie konstant oder veränderbar sind. In strikt funktionalen Sprachen sind alle Bindungen konstant, das Konstrukt der Zuweisung gibt es nicht. Hier gehe ich von einer hybriden Sprache mit Zuweisungskonstrukten aus. Deshalb ist es um so wichtiger, daß konstante Bindungen unterstützt werden. Jeder Versuch, diese durch Zuweisungen zu modifizieren, führt zu einer Fehlermeldung.

Für die Sprachkonstrukte des Objektsystems muß genau überlegt werden, welche definierenden Konstrukte zu konstanten und welche zu veränderbaren Bindungen führen.

[19] In JAVA gibt es keine Closures, wohl aber einen ironisch-inversen Sonderfall: die Lebensdauer von Klassenvariablen (das Analogon zu Modulbindungen) ist im allgemeinen nicht an die Programmausführung gebunden und kann schon vorher enden. Der Versuch eines Zugriffs führt dann zur erneuten Bindungserzeugung und -initialisierung mit möglicherweise fatalen Folgen für die abhängigen Programmteile, siehe Kapitel 8.4.1.

Ausgehend von den gesetzten Kriterien, insbesondere Robustheit und Effizienz, erhalten konstante Bindungen den Vorzug. Alle Konstrukte, die mit `define` beginnen, erzeugen konstante Modulbindungen:

- `define class`,
- `define generic` und
- `define function`.

Nur `define method` erzeugt keine neue Modulbindung, sondern erweitert die entsprechende generische Operation um eine weitere Methode.

4.5.2 Abschlußeigenschaften von Objekten

Neben Eigenschaften von Bindungen spielen gerade in objektorienierten Programmiersprachen die Abschlußeigenschaften von Objekten eine besondere Rolle. Zusammen bilden sie die Grundlage für entsprechende Analyse- und Optimierungsverfahren, um die Robustheit und Effizienz von Software-Systemen zu verbessern. Während EuLisp [Padget *et al.*, 1993], darauf gesetzt hat, Abschlußeigenschaften von Objekten durch entsprechende Analysen des Compilers abzuleiten [Kind und Friedrich, 1993], verfolgte man in DYLAN ein Konzept des expliziten Abschließens von Objekte für ihre Benutzung bzw. Spezialisierung außerhalb der Bibliothek (Gruppe von Modulen), in der sie definiert werden [Shalit, 1996, S. 131 ff.]. Dieses Konzept wird in DYLAN *Sealing* genannt. Dabei können nur solche Objekte (Klassen und generische Operationen) abgeschlossen werden, die über definierende Konstrukte wie `define class`, `define generic` und `define method` erzeugt wurden. Klassen und generische Funktionen, die explizit über die Operation `make` erzeugt werden, können nicht abgeschlossen werden.

Das Abschließen von Klassen bedeutet, daß sie außerhalb der Bibliothek nicht spezialisiert werden darf. Das Abschließen einer generischen Operation bedeutet, daß sie außerhalb der Bibliothek nicht um weitere Methoden erweitert werden darf. Zusätzlich können generische Operationen für bestimmte Bereiche abgeschlossen werden. Dann dürfen keine weiteren Methoden für diese Bereiche hinzugefügt werden. Die Bibliothek wurde in DYLAN als Bezug des Abschließens gewählt, weil man sie als separate übersetzbare Einheiten sieht.

Im Prinzip kann das Sealing-Konzept von DYLAN hier übernommen werden. Allerdings sollten die Abschlüsse meiner Meinung nach generell wirken und nicht nur außerhalb einer Bibliothek, zumal ich Bibliotheken in meinem Entwurf nicht betrachte, sondern nur Module.

Auch JAVA unterstützt das Abschließen von Programmeinheiten [Gosling *et al.*, 1996]. So können Klassen als *final* deklariert werden, was ihre Spezialisierung unterbindet. Damit sind automatisch auch alle Methoden einer abgeschlossenen Klasse abgeschlossen, da es keine Methoden außerhalb der syntaktischen Klammer einer Klassendefinition geben darf.

Ein ebenso wichtiger Abschlußaspekt von Objekten ist die Konstanz ihrer Felder. Auch hier können die Konzepte von DYLAN bzw. JAVA übernommen werden.

Sprachkonstrukte, die explizit die Abschlußeigenschaften von Programmen betreffen, erlauben dem Programmierer gezielt den Grad der Offenheit für Erweiterungen seiner Programmbausteine zu wählen. Sie erlauben dem Sprachimplementierer auf transparente Weise die zulässigen Optimierungen zu bestimmen. Im folgenden Kapitel werden im Zusammenhang mit den Implementierungstechniken auch Fragen der Optimierung behandelt. Dabei werde ich die Abschlußeigenschaften aus Implementierungssicht diskutieren.

4.6 Zusammenfassung

In diesem Kapitel habe ich die wesentlichen Konzepte objektorientierter Programmierung unter Berücksichtigung ihrer systematischen Erweiterbarkeit bzw. Spezialisierbarkeit entworfen. Abbildung 4.10 zeigt die verschiedenen Sprachaspekte im Überblick.

Abbildung 4.10: Sprachaspekte im Überblick.

Ein Hauptmerkmal objektorientierter Programmiersprachen muß die Unterstützung der Klassifikation bzw. des Generalisierens und Spezialisierens der Struktur und des Verhaltens von Objekten sein. Entsprechende Vererbungskonzepte bilden daher einen Schwerpunkt dieser Arbeit. Mit der vorgestellten Mixin-Vererbung liegt ein Resultat vor, das die Vorteile

einfacher und multipler Vererbung nutzt und die jeweiligen Nachteile weitgehend vermeidet. Die Richtlinie, zwischen wesentlichen und zusätzlichen Aspekten von Objekten zu unterscheiden, sie textuell getrennt zu notieren sowie die Wechselwirkungen zwischen verschiedenen Aspekten nur auf eine explizite Art und Weise zuzulassen, trägt auch deutlich zu einem besseren Programmierstil bei.

Das entworfene Objektsystem ist nicht endgültig auf eine "dogmatische" Linie fixiert. Es ist offen für neue Ausdrucksmittel und unterstützt ihre einfache und systematische Realisierung durch die Bereitstellung eines Spracherweiterungsprotokolls.

Die strikte Unterscheidung zwischen Sprachkonstrukten der Objekt- und der Metaobjektebene sowie die Modularisierung des Spracherweiterungsprotokolls in Teilprotokolle verbessert den Programmierstil und sie ermöglicht und unterstützt die Modul- und Applikations-Kompilation unter Verwendung traditioneller Compilerbautechniken. Konstante Modulbindungen bei der Definition von Klassen und generischen Funktionen, die Eindeutigkeit von Methodendefinitionen sowie die garantierte Integrität von Metaobjekten liefern darüberhinaus die notwendigen Abschlußeigenschaften. Dies ist nur unter der Voraussetzung möglich, daß Redefinieren aus dem Standardverhalten herausgenommen und in speziellen Erweiterungsmodulen bereitgestellt wird. Auf diese Weise ist insgesamt die gleiche Funktionalität und Ausdruckskraft gegeben wie in CLOS, allerdings unter Berücksichtigung des Verursacherprinzips. Dabei gelingt es, sowohl die Effizienz als auch die Robustheit des Objektsystems signifikant zu verbessern.

In den folgenden Kapiteln werden nach der allgemeinen Diskussion der Implementierungstechniken drei konkrete Objektsystem vorgestellt, die schrittweise zur Umsetzung der hier beschriebenen Konzepte führen und in ΤΕΛΟΣ schließlich ihren Abschluß finden. Bezogen auf die konkreten Objektsysteme gehe ich auch auf exemplarische Spracherweiterungen ein.

Kapitel 5

Implementierungstechniken und Optimierungen

The only applicable research method is to accumulate experience by implementing a system, synthesize the experience, think for a while, and start over.

Sandewall, Erik 1984. Programming in an Interactive Environment: The LISP Experience. In: Interactive Programming Environments, McGraw-Hill, S. 33.

In diesem Kapitel gebe ich zunächst einen Überblick über die verschiedenen Implementierungsalternativen und vertiefe anschließend den von mir gewählten Weg der portablen und inkrementellen Spracherweiterung nach dem sogenannten Einbettungsmodell [Christaller, 1987]. Aber auch Techniken des traditionellen Compilerbaus beeinflussen den Sprachentwurf signifikant, weil sie wesentlich weiterreichende Programmoptimierungen erlauben. Insbesondere wird Modul- und Komplettkompilation ermöglicht und unterstützt. Dies ist für Sprachen der Lisp-Familie ein Novum.

Die Herausforderung besteht darin, so viele Optimierungen zu ermöglichen, wie notwendig, ohne die relevanten Ausdrucksmittel eines dynamischen Objektsystems mit einem Metaobjektprotokoll zu verlieren. Kosten und Nutzen von Sprachmitteln müssen in ein ausgewogenes Verhältnis unter Berücksichtigung des Verursacherprinzips gebracht werden.

5.1 Überblick

Zunächst gebe ich einen Überblich über die grundlegenden Alternativen der Implementierung von Programmiersprachen und diskutiere insbesondere ihre Vor- und Nachteilen bei der Implementierung von Objektsystemen.

5.1.1 Interpretation vs. Kompilation

Die Implementierung von Programmiersprachen beinhaltet in der Regel zwei Aufgaben: die Realisierung eines Compilers, der syntaktisch korrekte und semantisch zulässige Program-

me der Quellsprachen in äquivalente Programme einer zu wählenden Zielsprache transformiert sowie eines speziellen Laufzeitsystems, falls das Laufzeitsystem der Zielsprache für die Resultate der Übersetzung ungeeignet ist. Letzteres ist dann der Fall, wenn der Zielsprache ein anderes Verarbeitungsmodell zugrunde liegt als der Quellsprache. Für die Ausführung von Programmen höherer Programmiersprachen auf physikalischen Maschinen erfolgt die Übersetzung in der Regel über mehrere Zwischensprachen bis man schließlich beim Befehlssatz der Maschine angekommen ist.

Eine Alternative zum Übersetzen in eine Zielsprache ist die direkte Interpretation von Programmen der Quellsprache mit Hilfe eines Interpretierers (engl. *interpreter*). In welcher Sprache der Interpretierer implementiert ist, spielt im Prinzip keine Rolle. Oft kann der Interpretierer für eine höhere Programmiersprachen am besten in dieser Sprache selbst implementiert werden. Dies gilt auch für die Implementierung von Übersetzern. Dabei ist dann am Anfang das sogenannte *Bootstrap-Problem* zu lösen. Der gravierende Nachteil von Interpretierern ist ihre im Vergleich mit Compilern geringere Effizienz. Optimierende Programmtransformationen würden, da sie zur Ausführungszeit stattfinden müßten, auch zu lange dauern. Zu den Pluspunkten von Interpretierern zählen der geringere Aufwand, sie zu realisieren, sowie ihre idealen Voraussetzungen für die interaktive Programmentwicklung.

Welche Alternative man wählt, hängt nicht zuletzt von der Art der Quellsprache ab. Für Programmiersprachen mit ausgeprägten statischen Eigenschaften wird man eher einen Compiler realisieren, um schon vor der Programmausführung möglichst viele Analysen durchzuführen, z. B. Typprüfungen von Variablen. Für dynamische Sprachen mit optionaler Typisierung von Variablen liegt es nahe, einen Interpretierer und komfortable Debugger bereitzustellen.

Natürlich kann man auch beide Konzepte kombinieren. Hierbei sind die sogenannten *Bytecode-Compiler* bzw. *Bytecode-Interpretierer* zu nennen [Kind, 1998]. Ein Bytecode-Compiler übersetzt die Quellsprache in eine Bytecode-Sprache, die speziell für die jeweilige Quellsprache zugeschnitten ist und die sich effizienter interpretieren läßt als die Quellsprache. Typische Beispiele für diesen hybriden Ansatz sind Systeme für die Sprachen Smalltalk, JAVA und natürlich EULISP. Insbesondere kann man mit diesem Ansatzt ein hohes Maß an Portabilität der Anwendungen erreichen, weil z. B. ein in C implementierter Bytecode-Interpretierer auf fast allen Rechner- und Betriebssystem-Plattformen lauffähig ist.

Einen weiteren Ansatz, Interpretation und Compilation zu verbinden, stellen sogenannte *inkrementelle Compiler* dar. Hier verkleinert man die Übersetzungseinheiten bis auf die Ebene von Ausdrücken, so daß für den Programmierer der Eindruck entsteht, man würde mit einem Interpretierer arbeiten. Tatsächlich wird jedoch jeder auszuführende Ausdruck zuerst übersetzt, möglicherweise bis auf die Ebene der Maschinenbefehle. Anschließend wird das Übersetzungsergebnis dynamisch gebunden und ausgeführt. Die reinen Ausführungszeiten sind dann vergleichbar mit Maschinencode-Compilern. Hinzu kommt die Zeit für die Übersetzung des Ausdrucks.

5.1.2 Schichtenmodell vs. Einbettungsmodell

Bei der Abbildung einer Sprache in eine andere werden zwei Modelle unterschieden [Christaller, 1987, S.39 ff]. Das *Schichtenmodell* geht von einer strikten Trennung beider Sprachen aus. Alle Konstrukte der Quellsprache müssen in solche der Zielsprache übersetzt werden. Das *Einbettungsmodell* sieht eine Erweiterung der Zielsprache um neue Konstrukte der Quellsprache. Die meisten Konstrukte der Zielsprache werden von der neuen Sprache unverändert übernommen. Die neuen Sprachkonstrukte werden mit Hilfe der Basissprache (Zielsprache) implementiert. Voraussetzung ist natürlich, daß die Basissprache ausdrucksstark genug ist, um die neuen Konstrukte effizient zu realisieren. Beide Modelle haben ihre Vor- und Nachteile.

Die strikte Trennung der Quellsprache von der Zielsprache im Schichtenmodell sorgt dafür, daß die Semantik der Quellsprache explizit festgelegt und beschrieben werden muß. Daraus lassen sich dann nachprüfbare Eigenschaften der Quellsprache ableiten. Dabei stehen potentiell beliebig große Quellcode-Einheiten als Eingabe für die Übersetzung, so daß auch globale Optimierungen leicht realisiert werden können. Einen Nachteil stellt der beträchtliche Aufwand dieser Methode dar. Dies schließt auch die Notwendigkeit ein, fast alle Entwicklungswerkzeuge für die Quellsprache neu implementieren zu müssen.

Umgekehrt macht die fehlende explizite Trennung der Quell- und Zielsprache beim Einbettungsmodell es schwerer, eine saubere Semantikdefinition der eingebetteten Sprache sicherzustellen. Besondere Vorkehrungen müssen getroffen werden, um unerwünschte Brüche der Abstraktion zu vermeiden. Fehlt in der Basissprache die Möglichkeit, bestimmte Sprachkonstrukte auszublenden, so ist man auf die Einsicht der Benutzer der Quellsprache angewiesen, daß keine Implementierungsprimitive in ihren (höheren) Programmen verwendet werden. Am einfachsten läßt sich das Einbettungsmodell mit Hilfe von syntaktischen Erweiterungskonstrukten (Makros) realisieren. Diese unterstützen aber meistens nur eine lokale Eingabe, nicht zuletzt um Programme inkrementell entwickeln zu können.

Der große Vorteil des Einbettungsmodells ist die Einfachheit der Implementierung und die Möglichkeit fast alle Entwicklungswerkzeuge der Basissprache wiederzuverwenden, ohne sie neu implementieren zu müssen. Für komplexe wissensbasierte Systeme ist es ein besonderer Vorteil, daß man mehrere Wissensrepräsentationssprachen in ein und dieselbe Basissprache einbetten kann. Dadurch lassen sich für die jeweiligen Aufgaben die am besten geeigneten Sprachmittel bereitstellen, die zudem über die Basissprache miteinander kombiniert werden können. Ob die Basissprache interpretiert oder kompiliert wird, die Implementierung nach dem Einbettungsmodell profitiert von beidem gleichermaßen.

Sieht man ein Objektsystem als Teil bzw. als Erweiterung einer universellen Programmiersprache, so stellt das Einbettungsmodell für den experimentellen Teil meine Arbeit die bessere Alternative dar. Der Sprachentwurf berücksichtigt jedoch auch andere Implementierungstechniken. Im Rahmen des Projekts APPLY wurde insbesondere die Komplettkompilation realisiert [Bretthauer *et al.*, 1994]. Letztere soll daher im Vergleich mit inkrementeller und mit der Modulkompilation im nächsten Unterabschnitt diskutiert werden.

5.1.3 Inkrementelle, Modul- und Komplettkompilation

Die größte und kleinste Kompilationseinheit in Lisp bildete traditionell die Funktion, wie Abbildung 5.1 zeigt. So vorteilhaft dies für die schrittweise Programmentwicklung ist, so problematisch ist es im Hinblick auf die Generierung von schlüsselfertigen Applikationen. Heißt die Alternative *ausschließlich inkrementelle Kompilation*, so können Optimierungen, die über die einzelne Funktion hinausgehen, nicht durchgeführt werden. Auch Typprüfungen zur Übersetzungszeit können nur begrenzt stattfinden. Traditionell enthält die Lisp-Laufzeitumgebung auch die gesamte Entwicklungsumgebung oder Teile davon. Dafür sind inkrementelle Compiler natürlich am besten geeignet, weil zur Laufzeit in der Regel nur kleine Programmeinheiten dynamisch hinzukommen, nämlich Funktionen.

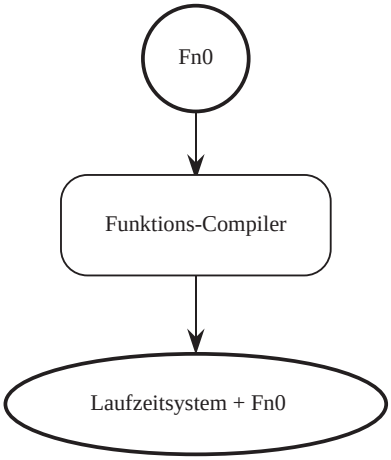


Abbildung 5.1: Inkrementelle Funktionskompilation

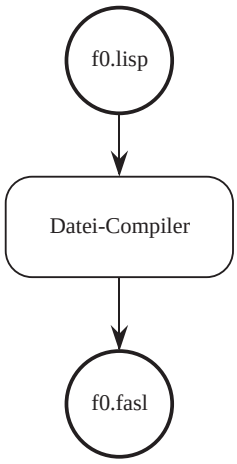


Abbildung 5.2: Datei-Kompilation

Da die Notwendigkeit für globale Optimierungen auch in Lisp offensichtlich war, hat man zunächst die *Datei-Kompilation* realisiert, siehe Abbildung 5.2. Dabei tragen jedoch Datei-

en keinerlei semantische Bedeutung. Sie können aus Programierersicht als Mittel zur manuellen Modularisierung von Programen verwendet werden, die Wechselwirkungen zwischen Dateien können jedoch nicht spezifiziert werden. Die einzelnen Dateien werden separat kompiliert, so daß keine übergreifenden Optimierungen durchgeführt werden können. Und selbst innerhalb einer Datei stehen Optimierungen auf wackeligen Füßen, weil es keinen syntaktischen und semantischen Abschluß gibt. Definitionen in einer Datei können durch solche in anderen Dateien erweitert und sogar überschieben werden, so daß datei-interne Optimierungen schnell obsolet werden können.

Wünscht man mittelgroße Kompilationseinheiten für Konsistenzprüfungen und Optimierungen, so braucht man ein Modulkonzept, wie es sich in anderen Sprachen bewährt hat. Wie in Abbildung 5.3 zu sehen ist, liest der *Modul-Compiler* nicht nur das zu übersetzende Modul als Eingabe, sondern auch die entsprechenden Definitionsdaten der Module, von denen das zu übersetzende Modul abhängt. Darin sind relevante Zusicherungen über Eigenschaften der exportierten Teile enthalten, die bei der Übersetzung dieser Module vom Compiler bereits abgeleitet wurden. Insbesondere lassen sich aus den Imports und Exports der Module auch einige Abschlußeigenschaften für Optimierungen ableiten.

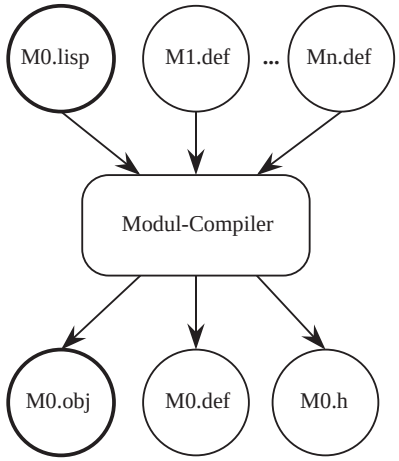


Abbildung 5.3: Modul-Kompilation

Das Ergebnis der Übersetzung eines Moduls ist also nicht nur der Zielcode, sondern auch Definitionsinformationen über die exportierten Programmelemente. Anschließend kann ein Modul samt allen Modulen, von denen es abhängig ist (das ist die transitive Hülle der Import-Relation), zu einer Applikation gebunden und ausgeführt werden.

Im nächsten Schritt können vollständige Applikationen als kompilierbare Einheiten betrachtet werden. Hier spricht man von *Komplettkompilation* oder auch *Applikationskompilation*. Wie in Abbildung 5.4 zu sehen ist, bilden alle Module, die zu einer Applikation gehören, die Eingabe für den Compiler, und zwar im Quellcode. Das Ergebnis der Kompilation ist dann ein ausführbares Programm (engl. *executable*).

Bei der Applikations-Kompilation können nun maximale Abschlußeigenschaften zuverlässig abgeleitet werden, weil von außen definitiv kein Einfluß auf die Interna einer Applikation ausgeübt werden kann. Je nach Gestaltung der Fremdfunktionsschnittstelle

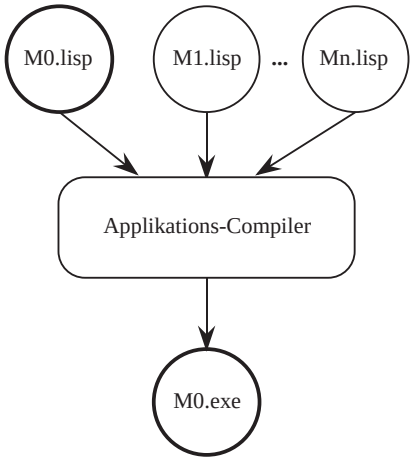


Abbildung 5.4: Applikations-Kompilation

(engl. *foreign function interface*) können nur noch Zeichenketten als Dateneingabe die Berechnungen beeinflussen.

Alle drei Arten, Funktions-, Modul- und Applikations-Kompilation habe ihre speziellen Stärken und Schwächen, je nachdem wofür sie verwendet werden. Idealerweise sollte eine ausgereifte Programmentwicklungsumgebung alle drei Kompilationsarten zur Verfügung stellen. Dabei kommt es auch darauf an, daß sie miteinander zusammenarbeiten können, so daß die Übergänge schrittweise erfolgen können. So können während der Bearbeitung eines Moduls Funktionsdefinitionen oder auch anderer Sprachkonstrukte innerhalb dieses Moduls inkrementell übersetzt und getestet werden. Einige Module können sich schon in einem stabileren Entwicklungszustand befinden, sie werden seltener modifiziert und können somit bei Bedarf als Einheit übersetzt werden. Und schließlich gibt es in der Regel auch viele Module, die kaum mehr verändert werden müssen, vielleicht auch nur von einem anderen Entwicklerteam geändert werden dürfen, und daher als Einheit bereits durch Applikations-Kompilation übersetzt vorliegen. Entsprechend unterschiedlich ist dann auch der Optimierungsgrad der Übersetzungsergebnisse. Fertige Applikationsteile sind über Modulgrenzen hochoptimiert, selten zu ändernde Module sind intern optimiert, verwenden jedoch nur die Schnittstellenbeschreibungen andere Module. Das aktuell zu entwickelnde Modul wird so übersetzt, daß die Edit-Compile-Test-Zyklen möglichst kurz sind.

5.1.4 Laufzeitsystem vs. Entwicklungssystem

Eine wichtige Entscheidung, die man bei der Implementierung eines Objektsystems treffen muß, ist die, inwiefern Aspekte der Programmentwicklung speziell unterstützt werden sollen. Diese dürfen aber nicht die gewünschte Semantik und Effizienz der Sprachkonstrukte zur Laufzeit generell beeinträchtigen. In dieser Arbeit unterscheide ich daher zwischen den Anforderungen zur Laufzeit und zur Entwicklungszeit. Im Konfliktfall wird die Entscheidung zugunsten der Laufzeiteigenschaften getroffen.

5.1.5 Optimierungen zur Übersetzungs-, Lade- und Laufzeit

Neben den bekannten Optimierungsverfahren aus dem Compilerbau für prozedurale und funktionale Sprachen, wie *Inline-Kompilierung*, *Restrekursionsauflösung* und *Last-Call-Optimierungen*, *Typinferenz* und *Speicherplatzoptimierung* [Bretthauer *et al.*, 1994, S. 47 ff] sollen vor allem spezielle objektsystembezogene Optimierungen diskutiert werden. Letztere können auf drei Aspekte konzentriert werden:

- die Erzeugung und Initialisierung von Objekten,
- die Feldzugriffsoperationen und
- den generischen Dispatch.

Dabei erschöpfen sich die Optimierungsmöglichkeiten nicht in statischen Analysen während der *Übersetzungszeit*, wie *Datenfluß-* und *Kontrollflußanalyse*. Auch dynamische Analysen zur *Lade-* und *Laufzeit* sind für dynamische Sprachen der Lisp-Familie möglich. Insbesondere erlauben inkrementelle Compiler, die Ergebnisse solcher Analysen in Form

von neu generiertem Code unmittelbar in die laufende Applikation hinzuzubinden und auch auszuführen. Natürlich wird das Laufzeitsystem dadurch komplexer. Ähnliche Techniken werden auch in den neuesten Java-Laufzeitsystemen eingesetzt, wenn sogenannte *Just-In-Time-Compiler* (JIT) integriert sind und dafür sorgen, daß der Bytecode vor der Ausführung in Maschinencode übersetzt wird. Siehe hierzu Abbildung 5.5.

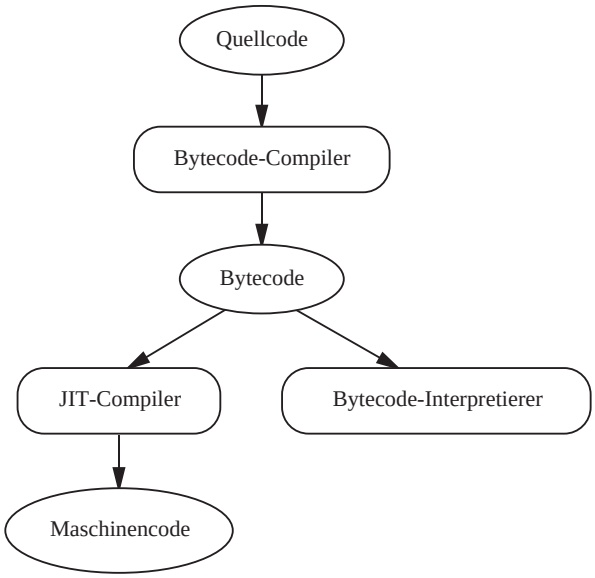


Abbildung 5.5: Just-In-Time-Kompilation

Ziel des Sprachentwurfs ist es, weder eine bestimmte Art der Implementierung noch einen bestimmten Zeitpunkt der Optimierungen zu erzwingen. Vielmehr sollte ein Spektrum von Implementierungstechniken ermöglicht und unterstützt werden. So interessant und effektiv Optimierungen zur Laufzeit sein mögen, insbesondere dann, wenn die statischen Analyseverfahren wenig Aufschluß über das zu erwartende dynamische Verhalten geben, so unangemessen sind sie im umgekehrten Fall. Welchen Sinn sollte es machen, Dinge, die zur Übersetzungszeit entscheidbar und optimierbar sind, bis zur Programmausführung aufzuschieben und Effizienznachteile in Kauf zu nehmen?

Es besteht auch ein umgekehrt proportionaler Zusammenhang zwischen dem Aufwand, der jeweils nötig ist, um die Berechnungen zu den anderen Zeitpunkten zu verkürzen. Will man die Laufzeiten maximal verkürzen, so müssen möglichst viele Optimierungen zur Übersetzungs- und Ladezeit erledigt werden. In CommonLisp lassen sich diese Phasen nicht völlig separieren, weil auch zur Laufzeit weitere Übersetzungen sowie Nachladen möglich sind. Das heißt, daß Übersetzen, Laden und Ausführen ineinander geschachtelt sein können.

Statische Optimierungen

Untersucht man, wie die verschiedenen Sprachkonstrukte sich auf die jeweilige Phase beziehen lassen, so stellt man fest, daß statische Definitionen von Klassen, generischen Operationen und Methoden weitgehend zur Übersetzungszeit analysiert und bzgl. der oben

genannten Aspekte optimiert werden können. Dies gilt insbesondere dann, wenn keine Spracherweiterungen der Metaobjektebene verwendet werden. Anderenfalls muß man zur Übersetzungszeit in der Lage sein benutzerdefinierte Module der Metaobjektebene, zumindest zum Teil, auszuführen. Diese Optimierungstechnik wird *partial evaluation* genannt. Die Idee ist dabei, daß man solche Ausdrücke, die zur Übersetzungszeit das gleiche Ergebnis liefern wie zur Lade- oder Laufzeit, durch das Auswertungsergebnis ersetzen darf. Dadurch können erhebliche Effizienzverbesserungen für die Programmausführung erreicht werden. Dies betrifft nicht nur die Laufzeit, sondern auch den Speicherbedarf, weil die Spracherweiterungsmodule unter günstigen Voraussetzungen während der Programmausführung nicht mehr benötigt werden. Solche Optimierungen können am besten durch die Applikations-Kompilation erreicht werden, weil dann am ehesten entschieden werden kann, ob die entsprechenden Optimierungsvoraussetzungen vorliegen oder nicht [Brett-hauer *et al.*, 1994, S. 47 ff]. Typinferenz zur Reduktion von Typprüfungen zur Laufzeit findet hier ihren Einsatz [Kind und Friedrich, 1993].

Dynamische Optimierungen

Eine entgegengesetzte Optimierungsstrategie besteht darin, Berechnungen erst dann anzustoßen, wenn ihre Ergebnisse benötigt werden. Dies wird mit *lazy evaluation* bezeichnet. So können umfangreiche Initialisierungen ggf. eingespart werden, wenn während einer konkreten Programmausführung bestimmte Pfade des potentiellen dynamischen Programmverhaltens gar nicht betreten werden. Zum Beispiel müssen Klassen nur dann initialisiert werden, wenn sie oder eine ihrer Subklassen auch instanziiert werden. Ähnlich verhält es sich mit dem generischen Dispatch. Eine generische Operation kann auf Objekte mehrerer Klassen anwendbar sein, tatsächlich kann es sein, daß sie nur auf Objekte genau einer Klasse angewandt wird. Entsprechend kann der generisch Dispatch auch gezielt auf diese Klasse hin optimiert werden.

Damit einher geht die Technik des *Caching*, wobei man immer wiederkehrende Berechnungen nur dann tatsächlich durchführt, wenn durch gewisse Änderungen ein anderes Ergebnis als bei der letzten Berechnung, deren Ergebnis man gespeichert hat, möglich wäre. Für generische Operationen lassen sich z. B. die anwendbaren Methoden für häufig vorkommende Klassen in kleinen Tabellen so speichern, daß sie sehr schnell gefunden werden [Kiczales und Rodriguez, 1990]. Gerade in dynamischen Sprachen wie Lisp sind diese Techniken sehr beliebt. Sie harmonieren am besten mit inkrementeller Kompilation. Nachteil dynamischer Optimierungen ist, daß sie besonders wertvolle Zeit kosten, nämlich Ausführungszeit.

Einen Kompromiß zwischen statischen Optimierungen zur Übersetzungszeit und dynamischen Opotimierungen zur Ausführungszeit stellen Optimierungen zur Ladezeit eines Programms dar. Welche Technik am angemessensten ist, hängt nicht zuletzt von der Art einer Applikation ab. Entsprechend konfigurierbare Compiler sowie Laufzeitsysteme, die das gesamte Spektrum möglicher Performanzoptimierungen abdecken, fehlen jedoch.

In den nächsten Abschnitten diskutiere ich speziell diejenigen Implementierungstechniken, die im experimentellen Teil dieser Arbeit verwendet wurden.

5.2 Voraussetzungen für die Einbettungs- und Erweiterungstechnik

Wie schon oben erwähnt, stellt die Einbettungs- und Erweiterungstechnik bei der Implementierung von höheren bzw. abstrakteren Programmier- und Wissensrepräsentationssprachen gewisse Anforderungen an die Basissprache, die erweitert werden soll. Einige Sprachkonstrukte und -Konzepte, wie Makros müssen von der Basissprache bereitgestellt werden, andere sind nützlich und wünschenswert, wie benutzerdefinierte Datentypen, aber nicht unbedingt notwendig. Schließlich gibt es einige Konstrukte, die sehr mächtig und bei Lisp-Programmierern beliebt sind, aber eher mit großer Vorsicht benutzt werden sollten, wenn überhaupt, z. B. die Funktion `eval` in CommonLisp, die oft die Hauptursache für schlechte Performanz darstellt.

5.2.1 Syntaktische Erweiterbarkeit: Makros

Die syntaktische Erweiterbarkeit ist die wesentliche Voraussetzung einer Programmiersprache, um noch höhere bzw. andere Sprachen in diese einzubetten. Natürlich kann man die Funktionalität einer Sprache erweitern, indem weitere Module, Klassen bzw. Bibliotheksfunktionen bereitgestellt werden. Dadurch ändert sich aber nicht die Art der Ausdrucksmittel. Eine rein prozedurale Sprache kann auf diese Weise nicht deklarativ werden. So wie in der Mathematik ständig neue Notationen eingeführt und verwendet werden, so muß man auch in einer universellen Programmiersprache in der Lage sein, neue syntaktische Konstrukte zu definieren und zu benutzen. Auf diese Weise können überflüssige Implementierungsdetails versteckt und alternative Implementierungsstrategien ausprobiert werden, ohne ständig neue Parser etc. schreiben zu müssen. Eine ausführliche Darstellung des Makro-Konzepts findet man in [Queinnec, 1996, S. 311 ff].

In Lisp gibt es traditionell zwei Arten von Makros, die nach dem Zeitpunkt ihrer Expansion unterschieden werden. Einlesemakros (engl. *read macros*) werden vom Parser expandiert, also während das Quellprogramm eingelesen wird, und beziehen sich auf ein einzelnes *Token*. Beispielsweise kann man ein Einlesemakro verwenden, um eine syntaktische Notation in Form eines dem Feldnamen vorangestellten Fragezeichens für lesende Feldzugriffe bereitzustellen:

```
(defun field-access-reader (stream char)
  `(cdr (assoc ',(read stream t nil t) field-values :test #'eq)))

(set-macro-character #\? #'field-access-reader)
```

Ein Vorkommen im Programmtext würde dann wie folgt ersetzt:

```
?x  ==>  (cdr (assoc 'x field-values :test #'eq))
```

Der Funktionsrumpf von `field-access-reader` besteht aus einem sogenannten *Backquote-Ausdruck*. Wird die Funktion angewandt, z. B. wenn der Lisp-Parser auf das Zeichen `?` stößt, so liefert sie das dem Backquote folgende Textmuster zurück, in dem die mit Komma beginnenden Teilausdrücke ausgewertet und durch ihre Ergebnisse ersetzt

wurden. In diesem Beispiel wurde angenommen, daß die Feldwerte in einer Assoziationsliste gespeichert werden und mit dem Feldnamen als Schlüssel zugegriffen werden können. Entscheidet man sich später für eine andere Lösung, muß nur die Funktion `field-access-reader` geändert werden.

Mit Einlesemakros können aber keine zusammengesetzten syntaktischen Sprachkonstrukte geschaffen werden, die neue Daten- oder Kontrollstrukturen realisieren. Letzteres erlauben sogenannte Textmakros oder einfach Makros, die erst nach der lexikalischen Analyse expandiert werden. Die Definition eines Textmakros erfolgt syntaktisch ähnlich der einer Funktion. Eine Anwendung eines Textmakros führt aber nicht nur zur Berechnung eines Ergebnisses, sondern das Ergebnis wird zunächst als neuer Programmtext an Stelle der Makroanwendung eingesetzt. Man spricht daher auch vom Expansionsergebnis. Kommt ein Compiler zum Einsatz, so werden Makros zur Übersetzungszeit expandiert. Zur Ausführungszeit wird dann das entsprechende Makro-Expansionsergebnis ausgeführt. Ein einfaches Beispiel für Textmakros ist das Konditional `if`, daß man in Lisp typischerweise auf das allgemeinere, aber nicht so gut leserliche Konditional `cond` abbildet:

```
(defmacro if (condition if-true if-false)
   `(cond
       ((,condition) ,if-true)
       (t ,if-false)))
```

Ein Vorkommen von `if` wird dann vom Compiler oder Interpretierer zunächst textuell ersetzt und das Ergebnis wird dann im umgebenden Programmkontext ausgeführt:

```
(if (= x 0)
  't
  (error "..."))  ==>

(cond
  ((= x 0) 't)
  (t (error "...")))
```

Zur Berechnung des Expansionsergebnisses steht dem Programmierer im Prinzip der gesamte Sprachumfang von Lisp zur Verfügung. Allerdings muß man sich darüber im klaren sein, daß die Makroexpansion in der Umgebung der Compiler-Ausführung erfolgt, während das Expansionsergebnis in der Umgebung des übersetzten Programms ausgeführt wird. Dieser Unterschied wird oft übersehen, wenn man mit einem Interpretierer arbeitet. Schlimmer noch, einige Interpretierer trennen die beiden Umgebungen nicht, so daß sich Programme interpretiert anders verhalten als kompiliert. Derartige Erfahrungen ermutigen einen nicht dazu, Makros zu benutzen. Für den Zweck der Spracherweiterung sind sie aber unverzichtbar.

Ein anderes bekanntes Problem mit Makros entsteht auf Grund der sogenannten *nicht-hygienischen Makroexpansion* [Kohlbecker *et al.*, 1986], [Bawden und Rees, 1988], [Clinger und Rees, 1991]. Dabei geht es darum, daß im expandierten Quellcode Namenskonflikte entstehen können, wenn die verwendeten Bezeichner vom Compiler nicht systematisch umbenannt werden. Auf der anderen Seite will man in gewissen Situationen gerade die

nicht-hygienische Makroexpansion ausnutzen, um z. B. `callNextMethod` zu implementieren. Darauf werde ich später noch zurückkommen.

Ohne auf den ewigen Streit über das Für und Wider Makros in der Lispgemeinde einzugehen, möchte ich mit den Worten von Christian Queinnec festhalten, worauf es eigentlich ankommt [1996, S. 311]:

> "While functions abstract computations and objects abstract data, macros abstract the structure of programs"

In diesem Sinne sollte jede moderne universelle Programmiersprache entsprechende Mittel bereitstellen, um neue problembezogene Abstraktionen der Programmstruktur definieren und benutzen zu können. Dies gilt um so mehr, wenn man wissensbasierte Systeme entwerfen und implementieren will.

5.2.2 Erweiterbarkeit der Datentypen

Will man ein Objektsystem nach dem Einbettungsmodell implementieren, so stellt sich die Frage, auf welchen Datentyp man die Objekte des Objektsystems abbilden kann. Im Prinzip kann dies ein beliebiger vordefinierter Typ sein, z. B. Liste oder Vektor. Ein Problem dabei ist, daß man den unerwünschten Durchgriff auf die innere Struktur so implementierter Objekte nicht verhindern kann. Dies beeinträchtigt die Robustheit und Zuverlässigkeit von Programmen. Ein weiteres Problem stellt sich im Zusammenhang mit der Notwendigkeit, Objekte des Objektsystems schnell zu erkennen bzw. von allen anderen unterscheiden zu können. Im Fall von Listen oder Vektoren ist der Test, ob es sich um ein Objekt des Objektsystems handelt, zu langsam. Insbesondere deshalb, weil er bei jedem generischen Dispatch durchgeführt werden muß.

Was man eigentlich braucht, ist ein neuer Vektortyp, der disjunkt von allen anderen Typen der Programmiersprache ist und dessen Zugriffsoperationen über den Index nach außen verborgen werden können. Weder CommonLisp noch Scheme bieten diese Möglichkeit. In CommonLisp kann man sogenannte Strukturen definieren, die disjunkt von allen anderen Typen sind und deren Felder wie Record-Felder behandelt werden. Somit kann man ein Feld für die Referenz auf die Klasse eines Objekts und ein weiteres Feld für den Vektor mir den Feldwerten des Objekts verwenden. In Scheme gibt es leider keine Strukturen, so daß man dort auf systemdefinierte Vektoren zurückgreifen muß, was mit den genannten Nachteilen verbunden ist.

Benutzerdefinierte Funktionstypen

Für die Repräsentation generischer Operationen bräuchte man einen eigenen Funktionstyp, damit man sie von allen anderen Funktionen unterscheiden und damit man in ihnen weitere Zustandsinformationen ablegen kann. Im Englischen werden solche Objekte *funcallable instances* genannt. Diese Möglichkeit bietet weder CommonLisp noch Scheme. Konkrete Lisp-Systeme haben zwar ihre implementierungsspezifischen Konstrukte dafür. Ihre Verwendung für die Implementierung meines Objektsystems hätte aber den Verlust der Portabilität zur Folge, so daß ich darauf verzichtet habe.

5.2.3 Zirkuläre Datenstrukturen

Zirkuläre Datenstrukturen braucht man, um den Kern der Metaobjektebene auf uniforme Weise zu implementieren. Jedes Objekt benötigt einen Verweis auf seine Klasse, ebenso die Klasse `Class`, die dabei auf sich selbst zeigt. Natürlich kann man die Zirkularität aufbrechen, indem man auf einen Namen oder eine Nummer verweist und in einer globalen Tabelle unter dem Namen oder der Nummer die eigentliche Klasse findet. Dies ist aber dann problematisch, wenn man auch anonyme Klassenobjekte erlauben will, die von der automatischen Speicherverwaltung freigegeben werden sollen, wenn kein Verweis auf sie mehr besteht. Werden sie in eine globale Tabelle eingetragen, können sie nie freigegeben werden.

So selbstverständlich es erscheinen mag, daß man in Lisp zirkuläre Datenstrukturen verwendet, um so ärgerlicher ist es, wenn konkrete Lisp-Systeme in Schwierigkeiten kommen, falls eine Liste oder ein Vektor sich selbst als Element enthält. Zum Beispiel:

```
(setf l (list 1 2 3))
(setf (car l) l)
```

Meistens tritt das Problem dann auf, wenn ihre externe Repräsentation ausgegeben wird. Oder aber wenn sie als konstante bzw. statische Objekte als Ergebnis der Kompilation in eine Datei geschrieben werden müssen.

5.2.4 Funktionen höherer Ordnung

Unter *Funktionen höherer Ordnung* werden solche Funktionen verstanden, die auf andere Funktionen als Argumente angewandt werden können. Anders betrachtet, verwendet man einen entgegengesetzt klingenden Begriff für denselben Sachverhalt: Man sprich von *Funktionen als Objekten erster Ordnung* und meint damit, daß Funktionen generell wie alle anderen Werte, also z. B. Zahlen oder Zeichenketten, behandelt werden und somit auch als aktuelle Parameter durchgereicht werden dürfen, um schließlich explizit angewandt zu werden. Diese Eigenschaft ist in Lisp eine Selbstverständlichkeit. Natürlich bringt sie eine Dynamik mit sich, so daß im Programm nicht überall mehr entscheidbar, welche Funktion an einer Stelle tatsächlich zur Anwendung kommt.

Im Prinzip bräuchte man für die Objektebene, wie sie in Kapitel 4 beschrieben wurde, keine Funktionen höherer Ordnung. Vom Konzept her wäre die Objektebene auch auf andere Programmiersprachen ohne Abstriche übertragbar. In Lisp spricht natürlich nichts dagegen, Funktionen wie alle anderen Objekttypen in das Objektsystem einzubeziehen.

Die Metaobjektebene nutzt allerdings explizit Funktionen höherer Ordnung, um z. B. Lese- und Schreiboperationen zu generieren und ihre Berechnung zu spezialisieren. Insofern stellen Funktionen höherer Ordnung auch auf der Implementierungsebene eine notwendige Voraussetzung dar.

5.2.5 Closures

Closures sind in Lisp Funktionen mit eigenem Zustand, weil sie bei ihrer Erzeugung, die lexikalische Umgebung einschließen. Da ihre Lebensdauer die Lebensdauer des sie

erzeugten Blocks überdauern kann, müssen auch die eingeschlossenen Bindungen weiter bestehen bleiben. Sie sind an die Lebensdauer der sie einschließenden Funktion gebunden. In Lisp können auch mehrere Funktionen dieselben Bindungen einschließen. Wird der Wert einer Variablen aus *einer* Funktion heraus verändert, so sind auch die *anderen* davon betroffen. Folgendes Beispiel aus CommonLisp soll dies verdeutlichen:

```
(defun create-closures (index)
  (list #'(lambda (vector) (svref vector index))
        #'(lambda () (setf index 0))))

(setf my-closures (create-closures 1))

(funcall (first my-closures) (vector 1 2 3))  ==>  2
(funcall (second my-closures))
(funcall (first my-closures) (vector 1 2 3))  ==>  1
```

Das Konzept der Closures ermöglicht es, dynamisch (also zur Ausführungszeit eines Programms) sich unterschiedlich verhaltende Funktionen zu erzeugen ohne zur Laufzeit jeweils neuen Quellcode generieren zu müssen, der dann erst übersetzt und hinzugebunden werden müßte. Mit anderen Worten stellen Closures einen gutartigen Weg dar, Code zur Laufzeit zu erzeugen und auszuführen, im Unterschied zu den Lisp-Funktionen `eval` und `compile` (siehe unten). Da das Funktionsmuster der Closures bereits zur Übersetzungszeit bekannt ist, muß zur Laufzeit nicht erneut übersetzt werden, sondern nur ein neues funktionales Objekt mit eigener Identität, der aktuellen Umgebung und dem gleichen Codemuster erzeugt werden. Diese Eigenschaft kann man ausnutzen, um Lese- und Schreiboperationen zu berechnen, die sich jeweils merken, auf welche Position in einem Vektor sie zugreifen müssen. Anderenfalls müßte zur Ausführungszeit einer Lese- bzw. Schreiboperation die entsprechende Position erst bestimmt werden. Das hätte signifikante Effizienznachteile.

Anonyme Lambda-Ausdrücke, *funargs* und Closures sind im Lambda-Kalkül von Anfang an gesehen worden [Weizenbaum, 1968]. Doch nur in Lisp haben sie auch eine sehr produktive praktische Bedeutung bekommen. Vom theoretischen Standpunkt kann man sagen, daß eine Programmiersprache mit Closures bereits objektorientiert ist. Der Zustand einer Umgebung wird funktional gekapselt, er ist von außen nicht einsehbar. Das funktionale Objekt reagiert, wenn man es aufruft in Abhängigkeit von den aktuellen Argumenten und vom Zustand der eingeschlossenen Umgebung. Nennt man das Aufrufen eines Closures ein *Sendeereignis* und die Argumente die *Botschaft*, so ist das Kommunikationsmodell perfekt. Die Möglichkeit, alle Objekte des Objektsystems als Closures zu implementieren wird im Abschnitt 5.4 diskutiert.

5.2.6 Interpretierer und Inkrementeller Compiler oder eval und compile

Nach weitverbreiteter Meinung ist es notwendig, einen Interpretierer oder einen (inkrementellen) Compiler zu Laufzeit einzusetzen, um reflektive Objektsysteme mit einem Metaobjektprotokoll zu realisieren. Dieses Vorurteil wird hier widerlegt. Natürlich kann es Vorteile haben, Quellcode explizit interpretieren bzw. kompilieren zu können. Setzt man diese Mittel gezielt ein, können sie einem eine Menge Arbeit ersparen.

Die Kunst besteht aber eher darin, zu erkennen, wann die Funktionen `eval` und `compile` fehl am Platz sind. Denkt man z. B. an hochoptimierte schlüsselfertige Applikationen, so sollten sie möglichst ohne `eval` und ohne `compile` auskommen. Der Interpretierer sollte nur dann zum Einsatz kommen, wenn tatsächlich die Notwendigkeit besteht, zur Laufzeit neuen Quellcode zu verarbeiten bzw. auszuführen. Der inkrementelle Compiler sollte nur dann zur Laufzeit verwendet werden, wenn er auch für entsprechende Randbedingungen konzipiert ist.

Leider kann der Aufruf der Funktion `compile` zur Laufzeit zu unvorhersehbaren Performanzproblemen führen, wenn z. B. die Prozessgröße plötzlich drastisch ansteigt und auch nach dem Abschluß des Compiler-Aufrufs nicht mehr kleiner wird. Natürlich sind dies in gewissem Sinne nur technische Mängel konkreter Lisp-Systeme, die nicht grundlegend gegen ihre Konzepte sprechen. Aus praktischer Sicht müssen in dieser Arbeit aber auch solche Probleme berücksichtigt werden.

Ein weiterer Grund, auf `eval` und `compile` zu verzichten, liegt darin, daß das Objektsystem selbst und auch seine Anwendungen komplettkompilierbar sein sollen. Die zur Verfügung stehenden Komplett-Compiler CLiCC und Eu2C unterstützen aber weder `eval` noch `compile` [Bretthauer *et al.*, 1994]. Geht man von der in CommonLisp spezifizierte Semantik dieser Funktionen aus, so sprechen prinzipielle Gründe gegen ihre Unterstützung durch Komplett-Compiler. Die Probleme liegen hier in der potentiell destruktiven Wirkung

- der Interpretation unbekannten (zur Übersetzungszeit) Quellcodes zum einen sowie
- der Generierung kompilierten Codes und seiner Ausführung zum anderen.

Es ist jedoch auch klar, daß z. B. die von CLiCC unterstützte Teilsprache CommonLisp0 ausdrucksstark genug ist, eine abgeschwächte Variante von `eval` zu implementieren - eine, die die Annahmen des Komplett-Compilers nicht umgehen und in der Laufzeitumgebung nichts zerstören kann. Entsprechendes gilt auch für eine semantisch unbedenkliche Variante der inkrementellen Kompilation zur Ausführungszeit. Will man also das Kind nicht mit dem Bade ausschütten, so kann man den Funktionen `eval` und `compile` eine Semantik geben, die mit der Komplett-Kompilation verträglich ist. Natürlich sind es dann nicht mehr die Funktionen aus CommonLisp.

5.3 Systemarchitektur und Bootstrapping

Bei der Diskussion objektorientierter Konzepte wurde zwischen der Objekt- und der Metaobjektebene unterschieden. Würde man zunächst nur die Sprachkonstrukte der Objektebene implementieren wollen, so könnte man diejenigen Konstrukte mit speziellen Auswertungsvorgaben als Makros realisieren und alle anderen als reguläre Funktionen. Die Gesamtheit dieser Makros und Funktionen kann man dann als einen partiellen Interpretierer objektorientierter Sprachmittel ansehen. Partiell deswegen, weil der größte Teil der Aufgaben vom zugrundeliegenden Lisp-System erledigt wird [Christaller, 1987, S. 40]. Der Begriff Interpretierer sollte hier aber nicht zu dem Mißverständnis führen, es müsse sich um Direktinterpretation objektorientierten Quellcodes handeln. In der Regel steht mit dem Lisp-System auch ein Compiler zur Verfügung. Es kommt nun bei der Implementierung des Objektsystems darauf an, vom Lisp-Compiler maximal zu profitieren.

Das Prinzip der Einbettungstechnik stellt sicher, daß objektorientierte Programme Lisp-Programme im herkömlichen Sinn sind und deshalb vom Lisp-Compiler übersetzbar sind. Das Ergebnis der Lisp-Kompilation kann im Idealfall so effizient sein, wie es auch eine nur lokal optimierende direkte Kompilation objektorientierter Sprachkonstrukte in Maschinencode nicht besser leistet. Der wesentliche Vorteil der Quellcode-Kompilation ist die Möglichkeit, globale Optimierungen vorzunehmen. Und es ist praktisch immer noch so, daß spezielle handcodierte Optimierungen des Laufzeitsystems auf der Maschinencode-Ebene (dies kann auch C sein) im Hinblick auf konkrete objektorientierte Konstrukte, wie z. B. generische Funktionen, zu höherer Effizienz führen.

Die Implementierung der Metaobjektebene ist komplizierter, weil sie inherent reflektiv ist. Soll auch die Implementierung diese Tatsache wiederspiegeln, steht man vor dem Henne/Ei-Problem. Was setzt man an den Anfang? Und wie schließt man den Zirkel? Die Lösungen des Problems werden schrittweise in den nächsten Kapiteln entwickelt, indem immer mehr Aspekte der Objektebene der Reflektion auf der Metaobjektebene unterstellt werden. Im Prinzip ist aber die Vorgehensweise in allen drei Systemen die gleiche. Sie besteht aus fünf Schritten:

1. Man definiert eine Schnittstelle elementarer Operationen zur Erzeugung von Objekten, einschließlich der primitiven Lese- und Schreiboperationen.
2. Man definiert vorläufige syntaktische und funktionale Konstrukte, um den minimalen Objektsystemkern, quasi in einem Schöpfungsakt, kreieren zu können.
3. In einer Bootstrap-Phase kreiert man manuell den minimalen Systemkern, bestehend aus den Kernklassen und einigen wenigen generischen Operationen sowie den erforderlichen Lese- und Schreiboperationen.
4. Anschließend können die endgültigen syntaktischen Sprachmittel als Makros implementiert werden, indem sie auf die Funktionalität des minimalen Kerns abgebildet werden.
5. Zum Schluß können die bisher offenen Metaobjektprotokoll-Teile objektorientiert implementiert werden, d. h. unter Nutzung des Objektsystems selbst.

5.4 Repräsentation von Objekten

Wie wir in Kapitel 4 gesehen haben, werden im Objektsystemkern im wesentlichen fünf Arten von Objekten unterschieden: Klassen, Felder, generische Operationen, Methoden und terminale Instanzen benutzerdefinierter Klassen. Da ich mich für das Einbettungsmodell als Implementierungstechnik entschieden habe, brauche ich mich zunächst nur um die neu hinzukommenden Objekttypen zu kümmern. Die meisten in Lisp schon vorhandenen Typen können vorausgesetzt und benutzt werden, insbesondere Listen, Vektoren, Funktionen, Symbole, Hashtabellen etc. Sie müssen nur in das Objektsystem derart integriert werden, daß sie wie benutzerdefinierte Klassen des Objektsystems benutzt werden können.

Die neuen Objekte sollen möglichst uniform repräsentiert werden. Dabei stellen sie eigene spezifische Anforderungen an die Repräsentation. Zum einen sollten alle möglichst wenig

Speicherplatz beanspruchen, zum anderen müssen die auf den Objekten häufig ausgeführten Operationen möglichst schnell sein. Der generelle Ansatz, um den Speicherbedarf zu minimieren, besteht darin, alles, was Objekte einer Klasse gemeinsam haben im Klassenobjekt zu halten. In den Instanzen einer Klasse sollten nur die wirklich instanzspezifischen Daten stehen. Daraus ergibt sich die Häufigkeit der Zugriffe vom Objekt zu seiner Klasse, der besonders optimiert werden muß.

Generell kann man zwei Arten der Objektrepräsentation unterscheiden. Bei der *funktionalen Objektrepräsentation* werden alle Objekte als Funktionen mit Zustand, also im Prinzip als Closures, implementiert[1]. Diese Lösung liegt vor allem dann nahe, wenn man vom Kommunikationsmodell ausgeht. Jedes einzelne Objekt wird als selbständig agierende Einheit gesehen, enthält also auch ein eigenes Programmstück, das es steuert. Geht man jedoch vom Konzept abstrakter Datentypen aus, so sind es nur die generischen Operationen und Methoden die als funktionale Objekte fungieren. Alle anderen Objekte, wie Klassen, Felder und terminalte Instanzen, können als passive *statische Objekte*, also z. B. Vektoren, repräsentiert werden.

Ohne es jedesmal explizit zu erwähnen, verwende ich im folgenden CommonLisp. Exemplarisch wird auf abweichende Lösungen in Scheme bzw. EuLisp hingewiesen.

5.4.1 Statische Objektrepräsentation

Wie bereits oben erwähnt, brauchen Objekte eine Hülle, die von außen erkennen läßt, daß es sich um ein definiertes Objekt der Erweiterungssprache und nicht um einen primitiven Wert, wie eine Zeichenkette oder eine Zahl, handelt. Gleichzeitig soll die Hülle verhindern, daß die innere Struktur eines Objekts von außen unkontrolliert eingesehen werden kann. CommonLisp bietet hierfür die Möglichkeit einen Strukturtyp (vergleichbar einem Record in anderen Sprachen) zu definieren:

```
(defstruct object
  ;; innere Struktur
  )
```

Bereits auf der Konzeptebene haben wir festgelegt, daß alle Instanzen einer Klasse die gleichen Felder besitzen und daß auf ihnen dieselben Operationen ausführbar sein sollen. In der inneren Struktur eines jeden Objekts müssen daher der Verweis auf die eigene Klasse sowie die eigenen Feldwerte untergebracht werden. Die Feldnamen und alle Feldannotationen können in der Klasse abgelegt werden. Natürlich gibt es in Lisp viele Möglichkeiten die Feldwerte zu repräsentieren, sie könnten

- in einer Liste mit den Feldwerten als Elemente der Liste,
- in einer Assoziationsliste als Name-Wert-Paare,
- in einer Eigenschaftsliste mit alternierenden Namen und Werten,
- in einer Hashtabelle mit dem Feldnamen als Schlüssel oder

[1] Im Prinzip deswegen, weil es einen Trick gibt, Datenstrukturen quasi als Konstanten in eine Funktion einzuschließen, ohne daß die Funktion zum Closure wird [Christaller, 1988].

- in einem Array bzw. einfachen Vektor[2] stehen.

Entsprechend der Datenstruktur ergeben sich unterschiedliche Zugriffsprimitive. Unter den genannten Randbedingungen bieten Vektoren offensichtlich die effizienteste Lösung, sowohl vom Speicherbedarf als auch von der Zugriffsgeschwindigkeit. Dies haben auch systematische Messungen ergeben. Würde man für jede Klasse einen eigenen Strukturtyp definieren, so könnte man auch die Felder der Klasse auf die sogenannten Slots des Strukturtyps abbilden. Dies hätte allerding zur Folge, daß man anonyme Klassen nur unter Anwengung von `eval` erzeugen kann, was unbedingt vermieden werden sollte. Und selbst wenn in Kapitel 4 auf der Objektebene keine anonymen Klassen vorgesehen wurden, sollte diese Möglichkeit in der Implementierung nicht ohne Grund verbaut werden.

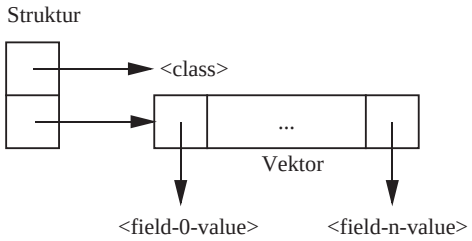


Abbildung 5.6: Statische Objektrepräsentation

Man kann die Objektrepräsentation nun graphisch, wie in Abbildung 5.6, darstellen oder mit folgender Lisp-Definition präzisieren:

```
(defstruct object
  class    ; die eigene Klasse
  fields)  ; Vektor mit den eigenen Feldwerten
```

Mit dieser Definition stehen einem automatisch ein Konstruktor, ein Typzugehörigkeits-Prädikat sowie Lese- und Schreiboperationen zur Verfügung:

```
(setf test-object (make-object :class 'dummy :fields (vector 1 2 3)))
(object-p test-object)       ==> T
(object-class test-object)   ==> dummy
(object-fields test-object)  ==> #(1 2 3)
(setf (object-class test-object) ...)
(object-class test-object)   ==> #S(object ...)
```

Lese- und Schreiboperationen

Um nun auf die Feldwerte zuzugreifen, muß man die Position eines Felds bestimmen. Mit der Initialisierung einer Klasse stehen alle Feldpositionen fest, so daß bei Generierung einer Lese- und einer Schreiboperation für jedes Feld ein ebenso schneller Zugriff gesichert ist, wie der Zugriff auf ein Vektorelement. Folgende Funktionen könnten benutzt werden, um für einen gegebenen Index die primitive Lese- bzw. Schreibfunktion zu erzeugen:

[2] In CommonLisp werden eindimensionale, nicht-verlängerbare Arrays Vektoren genannt; nicht zu verwechseln mit Vektoren in Java, die den Listen in Lisp entsprechen.

```
(defun make-primitive-reader (index)
  #'(lambda (object)
      (svref (object-fields object) index)))

(defun make-primitive-writer (index)
  #'(lambda (object new-value)
      (setf (svref (object-fields object) index) new-value))
```

Hier ermöglichen Closures eine elegante Realisierung der Zugriffsprimitive. Natürlich kann man auch einfache Funktionen, z. B. für die ersten 20 Feldpositionen, statisch definieren und für eine gegebene Position überall dasselbe Funktionsobjekt verwenden. Letztere Variante verbessert die Effizienz bzgl. Laufzeit und Speicher noch weiter. Bei entsprechenden Compiler-Optimierungen, inkl. Inline-Kompilation, können diese Lese- und Schreibprimitive auf eine einzige Maschineninstruktion reduziert werden.

Leider können solche primitive Funktionen im allgemeinen nicht direkt als Zugriffsoperationen der Objektebene verwendet werden. Das Hauptproblem stellt hierbei die multiple Vererbung dar. Erbt eine Klasse Felder von zwei Superklassen, so können zwangsläufig in der neuen Klasse nicht alle Feldpositionen aus den Superklassen übernommen werden. Die Lese- und Schreiboperationen der Superklassen sollen aber wie alle anderen Operationen auch auf Instanzen aller Subklassen ausführbar sein. Man wird also sorgfältig darauf achten müssen, ob eine Klasse nur als einzige oder als eine von mehreren Klassen spezialisiert werden kann. Im letzteren Fall, bietet es sich an, generische Lese- und Schreiboperationen zu verwenden, so daß für die jeweilige Klasse eine entsprechende Methode ausgeführt wird. In der jeweiligen Methode wird dann die oben beschriebene primitive Zugriffsfunktion ausgeführt.

Sollen die Lese- und Schreiboperationen typsicher sein, so müssen sie ggf. einen Typcheck durchführen. Verwendet man generische Operationen, so wird dies automatisch miterledigt. Anderenfalls muß der Typcheck der primitiven Zugriffsfunktion hinzugefügt werden. Da der Typcheck ggf. auch einen Subtypcheck erfordert, kann dies im allgemeinen (unter Annahme der multiplen Vererbung) *vier* Maschineninstruktionen kosten [Vitek *et al.*, 1997]. Vergleicht man diesen Aufwand mit nur *einer* Maschineninstruktion für den eigentlichen Zugriff, so wird deutlich, wie wichtig statische Analyse, Typinferenz und Typcheck-Elimination sind (siehe Abschnitt 5.5.1).

Vorkehrungen für das Redefinieren

Will man von vornherein das Redefinieren von Klassen und die automatische Anpassung ihrer Instanzen vorsehen, so würde man aus einem Objekt nicht direkt auf die Klasse verweisen, sondern auf eine Hülle, die die Klasse enthält:

```
(defstruct &object
  wrapper    ; die Hülle mit der eigenen Klasse
  fields)    ; Vektor mit den eigenen Feldwerten
```

Es könnte auch eine Zahl als Index für eine globale Tabelle sein, was aber dazu führen würde, daß auch anonyme Klassen von der automatischen Speicherverwaltung nicht mehr

freigegeben werden könnten (sie wären nicht mehr *garbage collectable*), weil sie in einer Tabelle eingetragen sind.

5.4.2 Funktionale Objektrepräsentation

Die funktionale Objektrepräsentation umhüllt den inneren Zustand eines Objekts mit einer Funktion. Dadurch ist es auch von außen nicht einsehbar. Das einzige, was man mit einer Funktion machen kann, ist, sie auszuführen. Nun stellt sich die Frage, welches Funktionsmuster jedes Objekt bekommen soll bzw. welche Aufgabe die Funktion über die Einhüllung des Objekts hinaus haben soll? Eine plausible Antwort ergibt sich im Kontext eines einstufigen Objektsystems ohne Klassen. Dabei ist jedes einzelne Objekt autonom, es hat seine eigenen Felder und Methoden. Wird es mit einem Nachrichtenschlüsselwort und ggf. weiteren Argumenten aufgerufen, so wählt es gemäß dem Schlüsselwort die passende Methode und führt sie, ggf. auf den durchgereichten Argumenten, aus. Folgendes Beispiel zeigt so ein handcodiertes Objekt:

```
(setf cons-cell
      (let ((left 1)                    ; Initialisierung der Felder
            (right 2))
        (let ((car #'(lambda () left))  ; Initialisierung der Methoden
              (cdr #'(lambda () right)))
          ;; die funktionale Hülle
          #'(lambda (selector &rest args)
              ;; message handler
              (case selector
                ((:car) (funcall car))
                ((:cdr) (funcall cdr))
                (t (error "No applicable method."))))))))

(funcall cons-cell :car)  ; ==> 1
(funcall cons-cell :cdr)  ; ==> 2
```

Dabei wird deutlich, unter welchen Voraussetzungen Closures, wie sie in CommonLisp oder Scheme zur Verfügung stehen, die angemessene Implementierung für Objekte darstellen. Alle Methoden müssen zum Zeitpunkt der Erzeugung textuell eingeschlossen werden. Anderenfalls können die Felder bzw. Objektvariablen nicht wie sonstige lokale Variablen verstanden und benutzt werden. Nachträglich können auf diese Weise keine neuen Methoden hinzukommen. Es wird auch klar, daß man auf diese Weise keine neuartigen Objekte zur Laufzeit erzeugen kann, ohne `eval` oder `compile` aufzurufen. Diese Einschränkung würde in zweistufigen Objektsystemen der Restriktion entsprechen, daß alle Klassen statisch definiert werden müssen.

Um diese Nachteile zu vermeiden, muß man nun das Herzstück der Closure-Lösung aufgeben, daß der Zustand von Objekten als lexikalische Umgebung einer Funktion repräsentiert wird und Instanz- bzw. Objektvariablen in Methoden wie reguläre Variablen in Lisp-Funktionen verwendet werden. Stattdessen wird man die Instanzvariablen bzw. Felder ähnlich repräsentieren wie bei der statischen Objektrepräsentation und die Methoden in

einem zweistufigen Objektsystem ebenfalls mit der Klasse assoziieren. Dann ergibt sich im Prinzip folgendes Muster:

```
(setf test-object
      (let ((class 'dummy)           ; Initialisierung der Klasse
            (fields (vector 1 2)))   ; Initialisierung der Felder
          ;; die funktionale Hülle
          #'(lambda (selector &rest args)
              ;; rufe klassenspezifisch die passende Methode auf
              )))
```

Um in Methoden auf die Feldwerte zuzugreifen, muß ihnen schließlich der Vektor mit den Feldwerten zugänglich gemacht werden, so daß man an der Stelle auch nichts gewinnt.

Zusammenfassend ziehe ich die Schlußfolgerung, daß für zweistufige Objektsysteme die statische Objektrepräsentation angemessener ist. Dabei blieb die Frage nach der Repräsentation von generischen Operationen noch unberücksichtigt. Diese werden nun im nächsten Abschnitt behandelt.

5.5 Realisierung generischer Operationen

Bei der Realisierung generischer Operationen stellt sich natürlich auch die Frage nach ihrer Repräsentation als Lisp-Objekt. Wie schon oben erwähnt, gibt es weder in CommonLisp noch in Scheme die Möglichkeit, auf portable Weise neue Funktionstypen zu definieren. Dennoch besteht die Anforderung, daß generische Operationen wie alle anderen Funktionen in Lisp bzw. Prozeduren in Scheme aufrufbar sein sollen. Sie haben jedoch auch einen Zustand, der sich z. B. durch das Hinzufügen neuer Methoden ändern kann. Da generische Operationen Objekte erster Ordnung sein sollen, muß jede auch Instanz einer Klasse sein. Man muß also mindestens den Verweis auf die Klasse, die Parameterliste bzw. den Bereich der Operation sowie die assoziierten Methoden repräsentieren.

Der Vorteil der Integration des Konzepts des Nachrichtenversendens und der Funktionsapplikation oder noch allgemeiner gesehen des objektorientierten und des funktionalen Programmierstils besteht vor allem darin, daß es an einer verwendenden Stelle im Programm irrelevant ist, ob eine Operation generisch, d. h. objektorientiert, oder nicht-generisch, d. h. herkömlich funktional, ist:

```
(defun f (x) ...)
(defgeneric g (x) ...)
(f (g 42))
```

Unterschieden werden muß nur dort, wo es darum geht, das Verhalten einer Operation zu spezialisieren, was eben nur für generische Operationen möglich ist. Ohne diese Integration mit der funktionalen Notation könnten generische Operationen genau so repräsentiert werden, wie alle anderen Objekte des Objektsystems. Die ausführbare Lisp-Funktion ist dann einfach ein Bestandteil des zusammengesetzten Objekts. Man müßte sie etwas umständlicher aufrufen, nämlich über ein spezielles Konstrukt, wie `funcall-generic-function`

bzw. `apply-generic-function`, was bis zu einem gewissen Grad wieder dem `send` Konstrukt aus FLAVORS entspricht. Diese Lösung wird im ersten Experiment verfolgt, das auf *Message-Passing* beruht. Dabei werden generische Operationen nicht explizit eingeführt. Sie manifestieren sich implizit bzgl. eines Methoden-Namen entlang den Vererbungsbeziehungen (siehe nächstes Kapitel).

Funktionen können in Lisp global oder lokal, benannt oder anonym sein. Wünschenswert ist es, auch für generische Operationen keine dieser Möglichkeiten zu verbauen.

```
(defun f (x) ...)                ; globale, benannte Funktion
(let ((fn #'(lambda (x) ...)))   ; lokale, anonyme Funktion
  ...)
```

Für die Objektebene des Objektsystems stellt die Beschränkung auf globale, benannte generische Operationen keine signifikante Einschränkung dar. Programme, die nur Sprachkonstrukte der Objekteben nutzen, sollen ja gerade statisch analysierbar sein. Für die Metaobjektebene, wie sie in Kapitel 4 beschrieben wurde, braucht man die Möglichkeit, generische Funktionen lokal und anonym zu erzeugen, um die Lese- und Schreiboperationen zu berechnen. Letztlich hängen Klassen und generische Operationen so miteinander zusammen, daß man anonyme generische Operationen benötigt, wenn man anonyme Klassen zulassen will. Beschränkt man sich darauf, alle Klassen statisch zu definieren, so könnte das Metaobjektprotokoll so gestaltet werden, daß eine statische Definition generischer Operationen auch akzeptabel wäre. Dadurch ließe sich die Implementierung des Objektsystems vereinfachen, insbesondere die Repräsentation generischer Operationen, womit der Faden dieses Abschnitts wieder aufgegriffen werden kann.

Wieder stellt sich die Frage, ob nicht Closures das passende Implementierungsmittel wären. Das Problem dabei ist, daß die eingeschlossene Umgebung von außen nicht zugänglich ist, jedenfalls nicht mit portablen Sprachmitteln. Nun könnte man den Trick anwenden, daß eine generische Funktion, implementiert als Closure, auf zwei verschiedene Weisen aufgerufen werden kann. Zum einen in ihrem eigentlichen Sinne auf solche Argumente, für die es auch entsprechende Methoden gibt. Und zum anderen ohne Argumente, um die Zustandsrepräsentation zu erhalten.

```
(defun make-funcallable-object (state)
  #'(lambda (&rest args)
      (if (null args)
        ;; liefere die innere Zustandsrepräsentation
        state
        ;; führe den generische Dispatch durch und
        ;; rufe die passende Methode auf
        ...)))

(setf f (make-funcallable-object (vector 1 2 3)))
(funcall f)        ==> #(1 2 3)
(funcall f 88 99)  ==> ein argumentspezifisches Ergebnis
```

Nachteil dieser Lösung ist der einhergehende Effizienzverlust für den regulären Aufruf generischer Funktionen. Ebenso nachteilig ist es, einen sogenannten *Rest-Parameter* verwenden zu müssen. Der Aufruf solcher Funktionen ist in der Regel langsamer als nur mit

geforderten Parametern. Darüberhinaus führen Rest-Parameter in den meisten Fällen dazu, daß bei jedem Aufruf mit Rest-Argumenten temporäre Listen alloziert werden und die automatische Speicherverwaltung dadurch zusätzlich belastet wird.

Will man eine portable Lösung, so bleibt schließlich nichts anderes übrig, als eine globale Tabelle für alle generischen Operationen anzulegen, in der dem ausführbaren Teil einer generischen Operation die Zustandsdaten zugeordnet werden.

```
(let ((generic-function-table (make-hash-tabel ...)))
  (defun find-generic-function (fn)
    (gethash fn generic-function-table))
  (defun add-generic-function (fn state)
    (setf (gethash fn generic-function-table) state)))
```

Man beachte, daß dabei keine globale Variable angelegt wird, die Tabelle bleibt hinter den Funktionen `find-generic-function` und `add-generic-function`, die in Wirklichkeit Closures sind, verborgen. Nachteil dabei ist, daß generische Funktionen nicht mehr von der automatischen Speicherverwaltung erfaßt werden können. Temporäre anonyme generische Operationen verbleiben daher in der Tabelle bis zur Beendigung des Programms. Sie können nicht vorher freigegeben werden. Dies könnte nur vermieden werden, falls es sogenannte *week pointer* in der Tabelle gäbe, die nur solange bestehen, wie es noch einen weiteren Verweis auf eine generische Operation gibt. Diese Möglichkeit steht aber weder in CommonLisp noch in Scheme zur Verfügung. Unter der Annahme, daß in typischen objektorientierten Programmen eher wenige temporäre generische Operationen erzeugt werden, hat die Portabilität an dieser Stelle Vorrang vor der Optimierung des Speicherbedarfs.

5.5.1 Generischer Dispatch

Bei der Implementierung des generischen Dispatchs muß zunächst geklärt werden, unter welchen Randbedingungen er durchzuführen ist. Auf der Objektebene sind alle Klassen, generische Funktionen und Methoden eines Moduls, einschließlich der importierten Module, zur Übersetzungszeit bekannt. Der Applikations-Compiler kann also unter der Annahme, daß keine weiteren Subklassen sowie Methoden hinzukommen, diverse Analysen, insbesondere Typinferenz, durchführen und die meisten Aufrufe generischer Funktionen so optimieren, daß nur noch wenige und einfachere Fallunterscheidungen übrigbleiben. Mit anderen Worten, der Laufzeitdispatch kann an vielen Stellen ganz eliminiert und an den übriggebliebenen Stellen optimiert werden. Der Modul-Compiler kann die obige Abschluß-Annahme nur für nicht exportierte Klassen und generische Operationen treffen. Exportierte Klassen und generische Operationen können in importierenden Modulen spezialisiert werden, so daß bestimmte Optimierungen zur Übersetzungszeit nicht zuläsig sind, oder aber sie müßten im tatsächlichen Spezialisierungsfall wieder rückgängig gemacht werden. Grundlage der Optimierungen beider Compiler sind globale Analysen.

Bei der Einbettungstechnik ist das Sichtfenster für potentielle Analysen auf das einzelne Sprachkonstrukt beschränkt. Dürfen Methodendefinitionen für eine generische Operation textuell verteilt sein, so muß die Optimierung des generischen Dispatchs inkrementell erfolgen. Auch die Implementierung der Objektebene erfordert somit ein Konzept wie für

die Metaobjektebene bzw. für voll dynamische objektorientierte Programmiersprachen, wo neue Klassen, generische Operationen und Methoden auch zur Laufzeit kreiert werden können. Daher muß auch der generische Dispatch in so einem Kontext dynamisch zur Laufzeit erfolgen. Hierfür liegen bereits verschiedene Optimierungskonzepte vor [Rose, 1988], [Kiczales und Rodriguez, 1990], [Rose, 1991], [Vitek und Horspool, 1994], [Queinnec, 1997],[Vitek *et al.*, 1997] auf die ich aber hier nicht weiter eingehen werde. Stattdessen stelle ich zwei ganz einfache Lösungen vor, die leicht zu implementieren sind und dennoch konkurrenzfähige Resultate erzielen, wie wir später sehen werden. Das Ziel dieser Arbeit besteht ja nicht darin, die dynamischen Optimierungen zu verbessern, sondern durch das entsprechende Sprachdesign grundsätzlich überlegenere statische Analyse- und Optimierungstechniken des traditionellen Compilerbaus zu ermöglichen. Gleichzeitig sollen die Konzepte auf einfache Weise mit der Einbettungstechnik effizient genug implementierbar sein.

Einfacher Methoden-Dispatch beim Message-Passing

Wie in Kapitel 3 schon ausgeführt wurde, kann das `send`-Konstrukt aus Objektsystemen mit Message-Passing direkt auf generische Operationen, die über *einen* Parameter diskriminieren, abgebildet werden. Die naheliegenden Implementierungsansätze unterscheiden sich aber deutlich. Bei zweistufigen Objektsystemen mit Message-Passing ist jede Methode *genau einer* Klasse zugeordnet. Auf Grund der Vererbung ist eine Methode nicht nur auf direkte Instanzen einer Klasse anwendbar, sondern auch auf die aller Subklassen. Gibt es entlang den Vererbungsbeziehungen mehrere definierte Methoden, so können bei einem konkreten Sendeereignis ggf. mehrere Methoden in einer bestimmten Reihenfolge zur Anwendung kommen. Läßt man kompliziertere Methodenkombinationsarten erst einmal außer Acht und beschränkt sich auf die Möglichkeit, aus spezielleren Methoden, die nächstallgemeinere aufzurufen (mittels `callNextMethod`), so liegt folgende Implementierung von `send` nahe:

```
(defun send (object selector &rest args)
  ;; usage: (SEND <object> <selector> <arg_{1}> ... <arg_{n}>)
  (let ((applicable-methods
         (find-applicable-methods (class-of object) selector)))
    (if (null applicable-methods)
      (error t
             "No applicable method: ~S in class: ~S."
             selector
             (class-of object))
      (apply-method (first applicable-methods)
                    applicable-methods
                    object
                    args))))
```

Der Methoden-Dispatch besteht also aus zwei Schritten:

1. der Suche nach den anwendbaren Methoden und

2. der Anwendung der speziellsten anwendbaren Methode falls vorhanden, und einer Fehlermeldung anderenfalls.

Um die anwendbaren Methoden bei gegebenem Selektor (Methoden-Namen) für eine gegebene Klasse möglichst schnell zu finden, kann man jeder Klasse eine Hashtabelle zuordnen mit Methodennamen als Schlüssel und den sortierten Listen aller anwendbaren Methoden als Eintrag. Dann muß die Suche nach allen anwendbaren Methoden in der Vererbungshierarchie jeweils nur einmal durchgeführt werden und das Ergebnis in der Hashtabelle eingetragen werden. Dies ist das übliche *Caching*. `find-applicable-methods` könnte wie folgt definiert werden:

```
(defun find-applicable-methods (class selector)
  (let ((cached-methods (fast-lookup (class-methods class) selector)))
    (if (null cached-methods)
      ;; slow case first time, must traverse class hierarchy
      (let ((new-entry (collect-applicable-methods class selector)))
        (unless (null new-entry)
          (add-to-cache (class-methods class) new-entry))
        new-entry)
      ;; fast (usual) case, just a hash table lookup
      cached-methods)))
```

Um den langsamen Fall zur Laufzeit ganz zu vermeiden, könnte man das Caching nach dem Laden eines Programms auch explizit entweder für alle Klassen oder selektiv, z. B. nur für die instanziierbaren Klassen, anstoen.

Die Methodentabelle kann man gleichzeitig auch für die definierten Methoden einer Klasse verwenden. Der Eintrag ist dann ein zusammengesetztes Objekt bestehend aus den direkt definierten Methoden und den Listen aller anwendbaren Methoden. Auf diese Weise wird sowohl für eine angemessene Anzahl von Hashtabellen als auch für angemessene Größen der Methodentabellen gesorgt. Als Faustregel kann man von durchschnittlich zehn Methoden pro Klasse ausgehen, so daß eine anfängliche Hashtabellengröße von sechzehn potentiellen Einträgen selten überschritten wird, aber auch nicht zu viel unnötiger Speicherplatz verbraucht wird.

Dispatch-Cache mit Assoziationslisten

Würde man nun das Konstrukt `send` auf generische Operationen abbilden, so wäre der eben skizzierte Implementierungsweg weniger angemessen. Man kann von durchschnittlich 1,5 Methoden pro generische Operation und etwa drei instanziierten Klassen, auf die eine generische Operation anwendbar ist, ausgehen. Daher würde eine Hashtabelle pro generische Operation mit der Klasse als Schlüssel viel zu viel unnötigen Speicher kosten.

Ist die Implementierungstechnik aus Objektsystemen mit Message-Passing schon für generische Operationen mit einfachem Dispatch nicht nutzbar, so muß für den mehrfachen Dispatch erst recht eine andere Lösung gefunden werden. Die grundlegende Technik ist wieder Caching. Wird eine generische Operation aufgerufen, so sucht man für gegebene

Argumente zunächst im Cache-Speicher nach den bereits sortierten anwendbaren Methoden. Falls man da keinen Eintrag findet, prüft man jede der assoziierten Methoden, ob sie anwendbar ist oder nicht. Falls mehrere anwendbar sind, müssen sie nach ihrer Spezifizität sortiert werden. Methoden werden hier jeweils mit der generischen Operation assoziiert, nicht mit einer Klasse. Wie sollte man auch bei mehreren Dispatch-Parametern eine Klasse zur Besitzerin der Methode erklären? Da eine generische Operation im Schnitt nur eine bis zwei Methoden hat, macht es keinen Sinn, dafür eine Hashtabelle anzulegen. Eine einfache Liste ist effizienter. Geht man von höchstens fünf Einträgen im Cache aus, so wäre auch hier eine Hashtabelle überdimensioniert. Stattdessen kann man eine Assoziationsliste verwenden, wobei die Klasse als Schlüssel fungiert. Wird nur über einen Parameter diskriminiert, ist die Assoziationsliste flach, bei Mehrfach-Dispatch ist sie geschachtelt:

```
((C1 . (m1 m0 ...))              ; Einfach-Dispatch
 (C2 . (m2 m1 m0 ...))
 ...)

((C1 . ((C2 . (m1 m0 ...))       ; Zweifach-Dispatch
         (C3 . (m0 ...))
         ...))
 (C2 . ((C1 . (m2 m1 m0 ...))))
 ...)
```

Die Dispatch-Funktionen können entsprechend optimiert werden, je nachdem über welchen Parameter diskriminiert wird. Die Berechnung des Diskriminators könnte z. B. für eine generische Operation mit zwei Parametern, wobei nur über den ersten diskriminiert wird, folgendermaßen aussehen:

```
(defun compute-dp1-p2-discriminator (methods)
  (let ((cache (list '())))    ; allocate new cache as closed variable
    ;; return a closure as a new discriminator
    #'(lambda (arg1 arg2)
        (let ((cached-methods  ; lookup for cached methods
               (cdr (assoc (class-of arg1) cache :test #'eq))))
          (if (null cached-methods)
            ;; bad case, slow lookup
            (let ((applicable-methods
                   (slow-lookup methods (class-of arg1))))
              (if (null applicable-methods)
                (error t "No applicable methods ..." ...)
                (progn
                  ;; add new entry to cache
                  (setf (rest cache)
                        (cons (cons (class-of arg1) applicable-methods)
                              (rest cache)))
                  ;; finally apply the first method
                  (funcall (first applicable-methods)
                           applicable-methods
```

```
                    arg1
                    arg2))))
          ;; good case, already cached
          (funcall (first cached-methods)
                   cached-methods
                   arg1
                   arg2))))))
```

Hier kann man sich auch eine Variante vorstellen, ohne Closures zu benutzen. Dann würde man entweder die generische Operation als Parameter oder die Methoden-Liste und die Cache-Assoziationsliste jeweils als Parameter der Diskriminator-Funktion durchreichen.

Inkrementelle Generierung von Dispatch-Funktionen

Steht ein effizienter inkrementeller Funktions-Compiler zur Verfügung, so kann man die Assoziationsliste auch in Quellcode, z. B. mit folgendem Muster gießen:

```
(lambda (arg1 arg2)
  (case (class-of arg1)
    ((C1) (case (class-of arg2)
            ((C2) (funcall m1 '(m1 m2 ...) arg1 arg2))
            ...))
    ...
    (t (error "No applicable methods ..." ...))))
```

Anschließend kann dieser Lambda-Ausdruck mittels der CommonLisp Funktion `compile` übersetzt und verwendet werden. Die Quellcode-Berechnung und seine Übersetzung dauern natürlich länger, als die Ergänzung einer Assoziationsliste. Auf der anderen Seite wird damit ein Minimum an Dispatch-Zeit benötigt. Daß dies eine sehr effiziente Lösung ist, bestätigt auch [Zendra *et al.*, 1997] für die Sprache Eiffel.

Wie aber oben schon ausgeführt, hat diese Lösung auch ihre Nachteile. Ist der inkrementelle Compiler zu langsam oder benötigt er zu viel Speicher, kann sie völlig inakzeptabel sein. Ebenso problematisch ist sie im Hinblick auf die Applikations-Kompilation eines so realisierten Objektsystems [Bretthauer *et al.*, 1994].

5.6 Sparsamer Speicherverbrauch

Im Zusammenhang mit der Objektrepräsentation und dem generischen Dispatch wurde schon die Speichereffizienz angesprochen. Dabei ging es vor allem um die Minimierung des mittel- bis langlebigen Speicherbedarfs. Klassenobjekte und Methodentabellen bzw. -assoziationslisten werden in der Regel für die Dauer der gesamten Programmausführung angelegt. Dies gilt auf jeden Fall für Programme, die nur die Objektebene nutzen: Klassen und Methoden sind statisch, d. h. zur Übersetzungszeit bekannt. Ein Applikations- oder ein Modul-Compiler kann den entsprechenden Speicher im statischen Bereich allozieren. Bei der Einbettungstechnik werden Klassen und Methoden im Sinne der Basissprache zwar

dynamisch angelegt, ihr Speicherbedarf geht aber als eine konstante Größe zur Ladezeit ein. Zur eigentlichen Ausführungszeit wird nur der Cache für den generischen Dispatch erweitert, aber auch hier gibt es eine Obergrenze. Im Prinzip könnte auch der Cache zur Ladezeit vollständig aufgebaut werden.

Kritisch ist vor allem das dynamische Speicherverhalten. Dieses wird maßgeblich durch die Instanzerzeugung und in Lisp auch durch die sogenannte *Cons-Zellen-Allokation* bestimmt. Auch wenn es eine automatische Speicherverwaltung (GC) gibt, gibt es sie nicht umsonst. Die Entbindung von manueller Speicherverwaltung entbindet nicht vom sparsamen Umgang mit Speicherressourcen. Dies wird häufig übersehen.

Neben dem Speicher für die direkte Instanzrepräsentation sollte möglichst kein temporärer Speicher bei der Instanzerzeugung alloziert werden. Ebensowenig darf während des generischen Dispatchs temporärer Speicher verbraucht werden. Dies kostet erstens Allokationszeit und zweitens Bereinigungszeit (GC). An zeitkritischen Stellen muß man daher sogenannte *Restparameter* vermeiden. Dies gilt nicht nur für die Einbettungstechnik. Auch Compiler-Analysen erlauben nicht immer, den entsprechenden Overhead zu eliminieren.

Als konkrete Schlußfolgerung ist festzuhalten, daß die Operationen `allocate` und `initialize` keine Restparameter haben sollen. Die Dispatchfunktionen müssen ebenso nur mit geforderten Parametern realisiert werden. Für häufig vorkommende Parameterkonstellation generischer Funktionen braucht man spezifische Dispatchfunktionen.

Folgendes Beispiel aus einer Referenzimplementierung von TELOS [Bradford, 1996], deren primäres Ziel die Validierung der Sprachspezifikation war, zeigt, wie man es nicht machen darf, wenn es auf Effizienz ankommt:

```
; cache, c-n-m
; cf compute-discriminating-function
; takes same args as the gf
(defun compute-primitive-discriminating-function (gf lookup-fn)
  (let* ((cache (generic-function-cache gf))
         (domain (generic-function-domain gf))
         (nargs (length domain)))
    #'(lambda (&rest values)
        (check-nargs gf (length values) nargs)
        (let ((applicable (cache-lookup
                           values
                           (discriminating-domain values domain)
                           cache
                           lookup-fn)))
          (if (null applicable)
              (error-no-applicable-methods gf values)
              (apply (car applicable)        ; apply-method-function
                     (cdr applicable)
                     values
                     values))))))
```

Die Performanzmessungen zu TELOS, siehe Kapitel 8.2.6 und Anhang 8.28, zeigen, welche Effizienzsteigerung spezielle Dispatchfunktionen bringen. Eine Faustregel lautet also: Keine Restparameter verwenden und möglichst mit `funcall` statt mit `apply` zu arbeiten, weil

die Funktion `apply` mit einem Restparameter definiert ist und bei ihrem Aufruf temporäre Listen alloziert werden.

Diese Richtlinien haben aber auch ihre Grenzen. Will man bei der Instanzerzeugung optionale Initialisisrungsschlüsselwörter in beliebiger Reihenfolge unterstützen, so kommt man nicht dran vorbei, sie in einer temporären Datenstruktur zwischenzuspeichern. Deshalb ist es angemessen für die Einstiegsoperation `make` bzw. `make-instance` einen Restparameter vorzusehen. Sonst müßte man für die Initialisierungsargumente explizit eine Datenstruktur erzeugen, z. B. eine Liste. Dies wäre aber ineffizienter, weil die implizite Allozierung von Restparameterlisten vom Compiler besser optimiert werden kann. Bei den Folgestationen, wie `allocate` und `initialize`, die vom Benutzer seltener explizit aufgerufen werden, kann auf den Restparameter verzichtet werden. Stattdessen werden die Initialisierungsargumente hier an einen geforderten Parameter gebunden.

5.7 Vermeidung der Quellcode-Interpretation

Will man eine hohe Effizienz des Objektsystems erreichen, so muß jede direkte Interpretation von Quellcode zur Laufzeit unterbleiben. Dies gilt übrigens nicht nur für Objektsysteme, sondern grundsätzlich für alle Lisp-Anwendungen, auch wenn viele diese Erkenntnis nicht einsehen (wollen). Es gibt wohl kaum Programme in CommonLisp, die nicht hier und da die kleine, effektive Funktion `eval` aufrufen, was man im folgenden Beispiel aus einem realen Objektsystem, nämlich MCF, sehen kann:

```
(defun create-object (a_class init-plist)
  `(let ((new-object
          (make-empty-object (length (get-all-slots ',a_class)))))
     ,@(compute-init-plist 'new-object  a_class (kwote-second init-plist))
     (setf (get-slot new-object 'class) ',a_class)
     (send-if-handles new-object :init ',init-plist)
     new-object))

(defun make-instance (a_class-name &rest args)
  (eval (create-object (eval a_class-name) args)))

(make-instance 'C1 :x 88 :y 99)   ==> #<C1 instance: ...>
```

Dabei soll `make-instance` der generelle Laufzeitkonstruktor für terminale Instanzen sein.

Das ist es ja gerade, was Lisp von anderen Programmiersprachen unterscheidet: die Möglichkeit, Programme als Daten anzusehen, sie einfach als Listen zu erzeugen und explizit auszuführen. Dagegen möchte ich im Prinzip nichts einwenden. Doch der Preis, der häufig dafür bezahlt werden muß, ist zu hoch: Programme sind dadurch nicht komplettkompilierbar, das Laufzeitsystem aus der Entwicklung muß im Laufzeitsystem der Anwendung enthalten sein, etc. Eine einfache Frage, die vor Benutzung von `eval` immer zu stellen ist, lautet: Wird an dieser Stelle im Programm wirklich ein *neuer Algorithmus* zur Laufzeit konstruiert, der nicht schon zum Zeitpunkt der Programm-Übersetzung bekannt ist bzw. bekannt sein könnte? Lautet die Antwort "Nein", dann sollte man kritisch abwägen, ob

Effizienz an so einer Programmstelle *wirklich keine Rolle spielt* und ob dieses Programm nie zu einer eigenständigen Applikation (ohne die Lisp-Entwicklungsumgebung) kompiliert werden soll?

Eine ähnliche Betrachtung muß man mit der Funktion `compile` in CommonLisp anstellen, die, angewandt auf einen sogenannten *Lambda-Ausdruck* als Quellcode, eine kompilierte Funktion zurückliefert:

```
(defun compute-constructor (class-name field-names)
  (compile nil
           `(lambda (&key ,@field-names)
              (send (make-object :class ,class
                                 :fields (vector ,@field-names))
                    :init ,(compute-init-list field-names)))))

(setf make-C1
      (compute-constructor 'C1 '(x y))  ==> #<Compiled function: ...>
(funcall make-C1 :x 88 y: 99)           ==> #<C1 instance: ...>
```

Wie wir später sehen werden, kann `compile` gerade zur Laufzeit eingesetzt werden, um Optimierungen zu realisieren. Der Vorteil gegenüber der Verwendung von `eval` ist hier, daß zwar einmal eine teure Operation ausgeführt wird, anschließend aber eine sehr schnelle Funktion als Ergebnis zur Verfügug steht. Andererseits bleibt der Nachteil, daß das Laufzeitsystem der Anwendung auch den inkrementellen Compiler enthalten muß.

5.7.1 Abstimmung syntaktischer und funktionaler Sprachkonstrukte

Ob man bei der Implementierung eingebetteter Sprachen auf `eval` und `compile` verzichten kann und es stattdessen gelingt, die entsprechende Arbeit auf den Datei-, Modul- bzw. Applikations-Compiler (oder den entsprechenden Interpretierer) "abzuwälzen", hängt nicht zuletzt vom gelungenen Sprachentwurf ab. Folgende Regeln müssen beachtet werden:

- Sprachkonstrukte, die globale Bindungen erzeugen, dürfen im Programm nur auf Toplevel stehen (hier: `defclass` und `defgeneric`).
- Sprachkonstrukte, die zur Übersetzungszeit eingesammelt und für Optimierungszwecke kombiniert werden sollen, dürfen nur auf Toplevel stehen (hier: `defclass`, `defgeneric` und `defmethod`).
- Voll funktional verwendbare Sprachkonstrukte, die also als Argument von `funcall` oder `apply` auftauchen dürfen, müssen sorgfältig von den nur syntaktisch verwendbaren Sprachkonstrukten unterschieden werde.
- Teilausdrücke syntaktischer Sprachkonstrukte, die ausgewertet werden sollen, müssen *vor* ihrem Durchreichen an funktionale Sprachkonstrukte ausgewertet werden. Funktionale Sprachkonstrukte erhalten nur Auswertungsergebnisse von Ausdrücken, sie selbst werten keine Ausdrücke aus, denn dazu müßten sie `eval` aufrufen.

- Sprachkonstrukte, die zur Übersetzungszeit syntaktische Vereinfachungen zulassen, sollten als Makros implementiert werden dürfen. Sie sollten daher nicht voll funktional benutzbar sein.
- Sprachkonstrukte, die Quellcode erzeugen, sollten als Makros konzipiert und implementiert werden, so daß der Quellcode zur Makroexpansions- und somit zur Übersetzungszeit berechnet werden kann, der dann vom Datei-, Module- oder Applikations-Compiler gleich mit übersetzt wird.
- Ist Codeberechnung zur Übersetzungszeit nicht möglich, muß sie zur Laufzeit auf die Generierung von Closures oder einfachen Funktionen mit zur Übersetzungszeit bekanntem Code-Muster beschränkt werden.

Ich greife das obige Beispiel nochmals auf, um zu zeigen, wie dabei die aufgeführten Regeln verletzt werden:

```
1 (defun create-object (cl init-plist)
2   `(let ((new-object (make-empty-object (length (get-all-slots ',cl)))))
3      ,@(compute-init-plist 'new-object  cl (kwote-second init-plist))
4      (setf (get-slot new-object 'class) ',cl)
5      (send-if-handles new-object :init ',init-plist)
6      new-object))

7 (defun make-instance (a_class-name &rest args)
8   (eval (create-object (eval a_class-name) args)))

9 (make-instance 'C1 :x 88 :y 99)   ==> #<C1 instance: ...>
```

Beginnt man mit der letzten Zeile, so sieht man, daß die Funktion `make-instance` als erstes Argument einen Klassennamen, also ein Symbol, erwartet. In den meisten Fällen weiß man im Programm, welche Klasse man instanziieren will. Da der Name somit feststeht, muß seine Auswertung unterbunden werden. Das macht der syntaktische Ausdruck `quote` (abgekürzt `'`). In der Implementierung hat man sich aber offensichtlich dafür entschieden, Klassenobjekte an ihre Namen in der globalen Bindungsumgebung zu binden. Der Zugriff über den Klassennamen könnte also auch einfach über `symbol-value` erfolgen. Stattdessen wird ein wesentlich generelleres Konstrukt verwendet, nämlich der Quellcode-Interpretierer, der beliebige Lisp-Ausdrücke auswerten kann. Dabei macht schon der Parameter-Name deutlich: was da als aktuelles Argument ankommen soll, ist ein Name, kein beliebiger Ausdruck, der zu einem Klassenobjekt ausgewertet werden muß. Hier kann man auf einen Schlag die Verwendung vereinfachen und die Implementierung beschleunigen:

```
7 (defun make-instance (a_class &rest args)
8   (eval (create-object a_class args)))

9 (make-instance C1 :x 88 :y 99)   ==> #<C1 instance: ...>
```

Um es mit einer Metapher zu beschreiben: Warum soll man Ketten anlegen, die anschließend mühevoll gesprengt werden müssen? Also, weg mit `quote` und `eval`! Natürlich ist das keine semantikerhaltende Transformation. Im Gegenteil, es ist ein Eingriff in den Sprachentwurf und in die Implementierung.

Jetzt steht da aber noch ein zweites `eval`. Was ist seine Aufgabe? Nun, man sieht, daß `create-object` in Wirklichkeit eine Liste konstruiert und diese als Ergebnis zurückliefert, kein Objekt im Sinne des Objektsystems. Insofern ist die Namensgebung an dieser Stelle irreführend. Die Liste, die für den obigen Aufruf (erst zur Laufzeit) berechnet wird, repräsentiert ein Code-Stück mit folgendem Muster:

```
2 (let ((new-object (make-empty-object
                      (length (get-all-slots '<C1-OBJEKT>)))))
3a  (setf (get-slot new-object 'X) '88)
3b  (setf (get-slot new-object 'Y) '99)
    ...
4   (setf (get-slot new-object 'class) '<C1-OBJEKT>)
5   (send-if-handles new-object :init '(:X 88 :Y 99))
6   new-object)
```

Die Idee hinter dieser Lösung war offensichtlich, die Instanzerzeugung dadurch zu optimieren, daß man für jeden Aufruf von `make-instance` speziell berechnete Code-Stücke einsetzt, statt dies von einem generellen Algorithmus tun zu lassen, der z. B. in einer Schleife die Initialisierung der Felder vornimmt. Dies wäre eine sehr effiziente Lösung, wenn man solche Code-Stücke schon zur Übersetzungszeit berechnen und statt der Aufrufe von `make-instance` einsetzen würde. So aber wird der Quellcode zur Laufzeit langsam interpretiert, mit der Folge, daß die Instanzerzeugung zu langsam wird.

Will man wirklich eine effiziente Lösung, muß auch das zweite `eval` weg, und `make-instance` muß ein Makro werden:

```
(defmacro make-instance (class-name &rest initargs)
  (compute-let-exp (symbol-value class-name) initargs))

(make-instance C1 :x 88 :y (+ 11 88))
 => ZUR ÜBERSETZUNGSZEIT expandiert zu:
 (let ((new-object (make-empty-object N))))
   (setf (get-slot new-object 'X) 88)
   (setf (get-slot new-object 'Y) (+ 11 88))
   ...
   (setf (get-slot new-object 'class) C1)
   (send-if-handles new-object :init (list :X '88 :Y '99))
   new-object)

  => ZUR LAUFZEIT ausgewertet zu:
     #<C1 instance: ...>
```

Für diese Variante müssen allerdings einige Voraussetzungen erfüllt bzw. Einschränkungen hingenommen werden:

- Die Klassenobjekte müssen zur Übersetzungszeit bekannt und initialisiert sein.
- In Aufrufen von `make-instance` müssen die Klasse und die Initialisierungsschlüsselwörter bekannt sein.
- `make-instance` kann nicht als Funktion verwendet werden, d. h. es darf nicht als Argument von `apply` oder `funcall` auftauchen.

5.8 Zusammenfassung

Will man die Lücke zwischen dynamischen und ausdrucksstarken Objektsystemen auf der einen Seite und den eher statischen und compiler-orientierten Objektsystemen auf der anderen Seite schließen, so muß vom Sprachdesign dafür gesorgt werden, daß ein ganzes Spektrum von Implementierungstechniken anwendbar wird. Idealerweise sollte ein Programmentwicklungssystem drei Compilationsarten bereitstellen:

- Modul-Kompilation,
- Applikations-Kompilation und
- inkrementelle Kompilation.

Daß auch Sprachen der Lisp-Familie von der Modul- und Applikations-Kompilation profitieren können wurde im Projekt APPLY [Bretthauer *et al.*, 1994] nachgewiesen. Der Verzicht auf Quellcodeinterpretation und -kompilation zur Laufzeit reicht im Prinzip aus, um traditionelle Kompilationstechniken erfolgreich anzuwenden.

Der hier gewählte Weg der Einbettungstechnik nutzt die Dateikompilation von CommonLisp, die konzeptionell zwischen der inkrementellen und der Modul-Kompilation liegt. Aber er nutzt auch implizit die inkrementelle Kompilation, falls das CommonLisp-System grundsätzlich bei jeder Interpretation zunächst inkrementell übersetzt und dann das Übersetzungsergebnis ausführt. Auch bei der Einbettungstechnik muß die eingebettete Sprache somit nicht interpretiert werden, falls es für die Basissprache einen Compiler gibt. Es ist aber ratsam, diesen nicht explizit zur Ausführungszeit aufzurufen, weil es dann Probleme mit der Dateikompilation geben kann und die Applikations- und Modul-Kompilation gänzlich unmöglich wird.

Als notwendige Voraussetzung der Basissprache für meine Realisierung sind Funktionen als Objekte erster Ordnung und Closures zu nennen. Darüberhinaus sind Makros als ein Mechanismus für syntaktische Erweiterungen der Basissprache praktisch unverzichtbar, aber prinzipiell nicht unbedingt notwendig. Schließlich braucht man die Möglichkeit, einen neuen Typ der Basissprache zu definieren, um das Objektsystem sicher und robust einzubetten. Anderenfalls muß dies mit Vektoren simuliert werden [Lange, 1993].

Die speichereffizienteste Objektrepräsentation bieten typisierte Vektoren, was man in CommonLisp indirekt mit einer Struktur (entspricht einem Record) aus zwei Feldern erreicht. Ein Feld dieser Struktur verweist auf die Klasse des Objekts und das zweite Feld verweist auf einen Vektor, in dem die Feldwerte des Objekts repräsentiert werden. Feldnamen und -annotationen liegen im Klassenobjekt. Diese Objektrepräsentation ermöglicht

auch schnelle Lese- und Schreiboperationen auf Objekten. Objekte dürfen der automatischen Speicherverwaltung nicht entzogen werden, etwa dadurch, daß im Klassenobjekt Verweise auf alle ihre Instanzen gehalten werden. Wenn schon nicht unbedingt Klassen, generische Funktionen und Methoden, so müssen mindestens terminale Instanzen *garbage-collectable* sein.

Der generische Dispatch kann in einer portablen Implementierung in CommonLisp eine gewisse Schallmauer nicht durchbrechen. Die Einbettungstechnik wird an dieser Stelle einer vergleichbaren Implementierung auf Maschinensprachebene immer unterlegen bleiben. Dennoch erzielt die einfache Lösung mit Assoziationslisten bei weniger als ca. fünf Einträgen im Cache ein akzeptables Ergebnis.

Die effiziente Objekterzeugung und -initialisierung hängt im wesentlichen von konzeptionellen Festlegungen ab. Optionale Initialisierungsargumente in beliebiger Reihenfolge haben ihren Preis, den man aber in Kauf nehmen sollte. Ein Compiler kann Aufrufe von `make` mit zur Übersetzungszeit bekannten Argumenten optimieren, so daß dieser Overhead entfällt. Aber auch bei dynamischer Abarbeitung aller Schritte zur Laufzeit kann ein effizientes Resultat erzielt werden, wenn die Allozierung temporären Speichers minimiert wird (kein `&rest`, kein `cons`). Dies gilt auch für den generischen Dispatch.

Zuletzt ist der für Lisp wichtigste Punkt zu nennen. Will man eine vergleichbare Effizienz wie in compiler-orientierten Programmiersprachen erreichen, so darf es keine Quellcodeinterpretation zur Laufzeit geben (kein `eval`). Dies gilt erst recht für die Einbettungstechnik.

Im nächsten Kapitel wird im Zusammenhang mit der ersten Version von MCS noch deutlicher, welche Folgen für die Effizienz eines Objektsystems die eben diskutierten Design- und Implementierungstechniken haben.

Kapitel 6

Der Einstieg: MCS nach dem Modell des Nachrichtenaustauschs

I would assert there is no hope of getting our complex designs right the first time.

Frederick P. Brooks, Jr. 1993. Keynote address: Language Design as Design, The Second History of Programming Languages Conference (HOPL-II), [Brooks, 1996, S. 13].

In diesem Kapitel stelle ich das erste von mir realisierte Objektsystem MCS vor, das auf dem Kommunikationsmodell bzw. dem Konzept des Nachrichtenaustauschs beruht. Ausgangspunkt ist dabei das für die MIT-Lispmaschinen entwickelte System FLAVORS [Weinreb und Moon, 1981], [Moon, 1986]. Mein praktisches Ziel ist natürlich nicht, die Systemsoftware der Lispmaschinen zu verbessern, sondern den unter FLAVORS entwickelten Werkzeugen zur Konstruktion wissensbasierter Systeme auch auf anderen Hard- und Software-Plattformen zu akzeptabler Performanz zu verhelfen. Dieses praktische Ziel wird nicht durch eine besonders ausgeklügelte Programmierung erreicht, sonder hauptsächlich durch ein gründliches Verständnis objektorientierter Sprachkonzepte, gefolgt von ihrer einfachen, portablen Implementierung. Als konkrete Anwendung für MCS dient die an der GMD entwickelte KI-Werkbank BABYLON [Christaller *et al.*, 1989], siehe Kapitel 2.1.1. Konkrete Zielplattformen sind Unix-Arbeitsplatzrechner, Apple Macintosh und IBM-kompatible PCs mit der Programmiersprache COMMONLISP. MCS wird daher als portables COMMONLISP Programm nach dem Modell der Spracheinbettung implementiert.

Ich beginne dieses Kapitel mit der Diskussion von FLAVORS bzw. der darauf gründenden Objektsysteme Micro Flavor System MFS [Christaller, 1988], Werex Flavor System WFS [de Buhr und Friederich, 1987] und Micro Common Flavors MCF [di Primio, 1988] sowie von OBJVLISP [Cointe, 1987]. Anschließend stelle ich den ersten Entwurf und die Implementierung von MCS vor, präsentiere die Performanzmessungen für BABYLON als Anwendung und bewerte das Gesamtergebnis.

6.1 Diskussion von Flavors

In Kapitel 3.4.1 wurde bereits das Micro Flavor System MFS vorgestellt. Es enthält nur die wesentlichsten Sprachkonstrukte von FLAVORS und wurde aus didaktischer Sicht mit Hilfe der Einbettungstechnik in COMMONLISP implementiert. Im Rahmen des WEREX-Verbundprojekts wurde daraus bei ADV-ORGA das WFS entwickelt. Letzteres diente unter anderen als objektorientierte Basis für BABYLON. Im Sprachgebrauch und zum Teil in der Literatur wurde zwischen MFS und WFS nicht unterschieden und für beide der Name MFS benutzt. Da es aber zwei verschiedene Systeme sind, verwende ich in dieser Arbeit auch unterschiedliche Namen.

Doch zunächst möchte ich einen kurzen Überblick über die Sprachkonstrukte aus FLAVORS geben, die in BABYLON verwendet wurden.

6.1.1 BABYLON-Schnittstelle zu Flavors

Um von einem konkreten Objektsystem unabhängig zu werden, wurde für BABYLON eine abstrakte Schnittstelle zu einem FLAVORS-ähnlichen Objektsystem definiert. Ihre ausführliche Beschreibung findet man in [Walther *et al.*, 1989]. Hier möchte ich sie kurz erwähnen, weil die verwendeten Benchmarks in dieser abstrakten Syntax formuliert wurden. Im Prinzip ist diese Schnittstelle direkt aus einer Teilmenge der FLAVORS-Sprachkonstrukte hervorgegangen. Zur syntaktischen Unterscheidung hat man die abstrakten Sprachkonstrukte mit einem "$" versehen, also z. B. `def$flavor` statt `defflavor`. Um als Basis für BABYLON zu dienen, mußte das jeweilige Objektsystem diese Schnittstelle auf ihre eigenen Konstrukte abbilden, was für FLAVORS selbst natürlich trivial war. Auf die MCS-Abbildung komme ich in Abschnitt 6.5 zu sprechen.

Statt die formale Spezifikation der Schnittstelle aus [Walther *et al.*, 1989] zu wiederholen, gebe ich ein kurzes Beispiel in Abbildung 6.1, in dem die wesentlichen Sprachkonstrukte vorkommen.

Im Vergleich zu MFS (siehe Seite 49) kommen im wesentlichen *Default-Werte* für Instanzvariablen in der Flavor-Definition, *Demon-Methodenkombination* in der Methodendefinition und einige vordefinierte Methoden auf allen Instanzen wie `:init`, `:operation-handled-p`, `:send-if-handles`, `:describe` etc. hinzu.

6.1.2 Defizite vorhandener Objektsysteme

Das Hauptproblem der vorhandenen Objektsysteme lag in ihrer unzureichenden Effizienz für die Anwendung BABYLON. In WFS waren die Instanzvariablenzugriffe und das Versenden von Nachrichten besonders zeitkritisch. Bei MCF war die Instanzerzeugung zu langsam und das dynamische Speicherverhalten kritisch. PCL fiel durch den enormen Speicherbedarf für sich selbst als auch für Anwendungen auf, so daß ein Laden des gesamten BABYLON-Systems auf einem Apple Macintosh seinerzeit nicht möglich war [Walther *et al.*, 1989].

Eine erste Analyse der Systeme läßt auch die Hauptursachen der Effizienzprobleme schnell erkennen:

```
(def$flavor Graphical-Object
  ()
  ())

(defvar *graphical-objects* '())

(def$method (Graphical-Object :after :init) (init-args)
  (push self *graphical-objects*))

(def$method (Graphical-Object :display) ()
  (print "Graphical-Object"))

(def$flavor Point
  ((x 0) (y 0))
  (Graphical-Object)
  :gettable-instance-variables
  :settable-instance-variables)

(def$method (Point :move) (x-pos y-pos)
  (setf x x-pos)
  (setf y y-pos)
  ($send self :display))

(def$method (Point :before :display) ()
  (print "Displaying Point at x = ~S, y = ~S." x y))

(compile-$flavor-$methods Point)

(setf p1 (make-$instance 'Point :x 2))

($send p1 :display)
  ==> Displaying Point at x = 2, y = 0.
      Graphical-Object

($send p1 :operation-handled-p :move)
  ==> T

($send (make-$instance 'Graphical-Object) :send-if-handles :move)
  ==> NIL
```

Abbildung 6.1: Beispiel der Verwendung von FLAVORS-Konstrukten in BABYLON.

- Es wurde häufig die Funktion `eval` verwendet.
- Die Repräsentation von Feldwerten in Assoziations- bzw. Eigenschaftslisten läßt keine schnellen Zugriffe zu und benötigt zu viel Platz.
- Die funktionale Objektrepräsentation erschwert schnelle Instanzvariablenzugriffe.
- Die Objektrepräsentation mit Symbolen macht alle Instanzen permanent, sie entgehen der automatischen Speicherverwaltung.
- Die Verwendung von Hashtabellen für generische Funktionen in PCL ist enorm speicherplatzaufwendig.

Konzeptuelle Probleme liegen darin, daß Klassen und terminale Instanzen nicht uniform behandelt werden. Selbst wenn Klassen in MCF auch Objekte sind, werden sie nicht über die gleichen Methoden erzeugt [di Primio, 1988, S. 15]. Bis auf PCL sind die Objektsysteme nicht reflektiv bzw. nicht erweiterbar, sie bieten kein Metaobjektprotokoll an. Hier stellt das Konzept von ObjVLisp (siehe Kapitel 3.4.2) eine interessante Alternative. Seine Implementierung war aber nicht auf Effizienz ausgerichtet. Aus diesen Beobachtungen wurde die erste Version von MCS entworfen und implementiert.

6.2 Entwurf von MCS 0.5

Den Entwurf von MCS 0.5 beginne ich mit seiner Gesamtarchitektur, den Designkriterien und der methodischen Vorgehensweise. Anschließend werden die Sprachkonstrukte entworfen. Zunächst wird die Objektebene betrachtet, dann die Metaobjektebene und schließlich die Aspekte der Programmentwicklungsumgebung. Diese Strukturierung ist allerdings nur ein Beschreibungsmittel, sie kann wegen eines fehlenden Modulkonzepts in CommonLisp nicht semantisch unterstützt werden.

6.2.1 Ziele, Systemarchitektur und Vorgehensweise

Das Meta-Klassen-System MCS soll als ein erweiterbarer Objektsystemkern entworfen und implementiert werden. Die Sprachkonstrukte und ihre Implementierung sollen:

- minimal,
- reflektiv,
- erweiterbar,
- uniform,
- orthogonal,
- effizient und
- portabel

sein. Die Forderungen nach *Minimalität*, *Reflektivität* und *Erweiterbarkeit* sind dabei weniger im theoretischen Sinne, sondern eher aus der Sicht des angewandten Software-Engineerings zu verstehen. Im Objektsystemkern sollen die essentiellen Sprachmittel bereitgestellt werden. Die Gesamtfunktionalität des Objektsystems sollte möglichst objektorientiert, also mit den eigenen Sprachmitteln selbst beschrieben und implementiert werden. Erweiterungen des Systemskerns sollen durch das bereitgestellte Metaobjektprotokoll unterstützt werden.

Die *Uniformität* drückt aus, daß man alle Objekte, Klassen und ihre Instanzen, in gleichen Aspekten auch gleich behandeln soll. Dies hängt auch mit der Reflektivität zusammen. Klassen werden wie in ObjVLisp als Objekte repräsentiert, sie sind somit selbst Instanzen einer Klasse. Eine ausgezeichnete Klasse, hier `standard-class`, ist Instanz von sich selbst (siehe Kapitel 3.4.2) und bildet die Wurzel der Instanziierungshierarchie.

Orthogonalität meint, daß die Sprachkonstrukte in vernünftiger Arbeitsteilung organisiert werden sollen. Nicht jedes Konstrukt bzw. Objekt soll alles können, vielmehr soll durch Komposition aller Teilaspekte die gewünschte Ausdruckskraft der Sprache insgesamt erreicht werden.

Die *Effizienz* und *Portabilität* des Objeksystems soll durch Vereinfachung der Sprachkonstrukte auf die praktischen Erfordernisse sowie ihre schnörkellose Implementierung mit Hilfe der Einbettungstechnik in CommonLisp gesichert werden. Darüber hinaus sollte eine einfache Portierung von MCS in andere Lisp-Dialekte möglich sein.

Sprachumfang und -Funktionalität orientieren sich zwar an der BABYLON-Schnittstelle, die konkreten Konstrukte können jedoch durchaus abweichen, um konzeptuell keine Kompromisse eingehen zu müssen. Es genügt, wenn die BABYLON-Schnittstelle auf Konstrukte von MCS abgebildet werden können.

6.2.2 Sprachkonstrukte der Objektebene

Auf der Objektebene werden folgende Konzepte unterstützt:

- Klassen mit multipler Vererbung nach der Linearisierungsstrategie *depth-first-left-to-right-up-to-joins*.
- Slots mit spezialisierbaren Default-Initialisierungswerten und automatisch generierten Lese- und Schreibmethoden. Neben diesen gibt es eine allgemeine Zugriffsfunktion `slot-value`, die auch außerhalb von Methoden verwendet werden kann. Der Zugriff auf Slots in Methoden über ihren Namen, wie auf eine reguläre Lisp-Variable wird nicht unterstützt. Dafür gibt es ein explizites Zugriffsprimitiv. Alle Slots sind initialisierbar.
- Nachrichtenversenden mit Methodennamen als Nachrichtenschlüssel.
- Einfacher Methoden-Dispatch über die Klasse des Empfängerobjekts.
- Methodenkombination mit *Primär-*, *Before-*, *After-* und *Around-Methoden*.
- Reguläre Lisp-Objekte wie Listen, Zahlen, Symbole etc. werden nicht in das Objektsystem integriert, man kann ihnen keine Nachrichten schicken. Es können auch keine Methoden für die Lisp-Basistypen definieren werden.

Obwohl sich MCS 0.5 an der Funktionalität von FLAVORS orientiert, verwende ich die Begriffe *Klasse* für *Flavor* und *Slot* für *Instanzvariable*, die Syntax ist entsprechend. Ansonsten kann das Beispiel aus Abbildung 6.1 auch als Beispiel für MCS 0.5 gesehen werden.

Nachrichtenversenden

MCS 0.5 folgt dem Kommunikationsmodell als einem objektorientierten Paradigma insofern, als die Aktivierung von Methoden über das Versenden von Nachrichten erfolgt. Das allgemeine Sprachprimitiv hierfür ist `send-message`. Es kann an jeder Stelle eines Lisp-Programms, innerhalb und außerhalb von Methodenrümpfen, mit folgender Syntax aufgerufen werden:

`(send-message` *receiver method-name* $\{$*argument*$\}^*$`)`

Da `send-message` als Funktion spezifiziert ist, kann es auch voll funktional verwendet werden, also z. B. als Argument von `apply`. Als erstes, überprüft `send-message`, ob *receiver* ein Objekt im Sinne des Objektsystems ist. Falls ja, wird über seine Klasse nach Methoden mit dem Namen *method-name* gesucht, die im Erfolgsfall auf dem Empfängerobjekt und den Argumenten ausgeführt werden. Wird keine passende Methode gefunden, wird an *receiver* die Nachricht `:default-handler` mit dem Methodennamen und denselben Argumenten geschickt. Falls *receiver* kein Objekt ist, wird ein Fehler gemeldet.

Um den aufwendigen Objekt-Test bei mehrfachem Nachrichtenversenden an dasselbe Objekt nicht wiederholen zu müssen, steht die Funktion `send-fast` zur Verfügung. Für weitere Nachrichten an das Empfänger-Objekt innerhalb von Methoden, das dort als `self` referiert werden kann, wird das Makro `send-self` mit folgender Syntax bereitgestellt:

`(send-self` *method-name* $\{$*argument*$\}^*$`)`

Es verhält sich semantisch wie `send-fast`, kann aber noch effizienter implementiert werden. Ein Compiler kann natürlich durch statische Analyse herausfinden, welche Aufrufe von `send-message` im Programm optimiert werden können. Dann könnte auf `send-fast` und `send-self` verzichtet werden. Einfache Analysen in Methodenrümpfen können aber auch mit Hilfe der Einbettungstechnik realisiert werden.

Klassendefinition

Klassen werden auf der Objektebene mit dem syntaktischen Konstrukt `defclass` definiert, das als Makro implementiert wird. Auf der Metaobjektebene können Klassenobjekte wie alle anderen Objekte mit `make-instance` erzeugt werden. In der Implementierung der Objektebene und in ihrer kontrollierten Offenlegung (der Metaobjektebene) werden Klassen und ihre Instanzen entsprechend uniform behandelt. Bei einer Klassendefinition gibt man den Namen der neuen Klasse an, ihre Slots und ihre direkten Superklassen. Optional kann man angeben, welcher Klasse die neue Klasse als Instanz gehört. Dies drückt die Klassenoption `:metaclass` aus. Die Default-Metaklasse ist die Klasse `standard-class`. Als Ergebnis der Klassendefinition wird im Unterschied zu FLAVORS ein neues Klassenobjekt an den angegebenen Klassennamen gebunden. Eine Klasse ist somit ein Objekt erster Ordnung und kann über ihren Namen, wie auch sonstige Lisp-Objekte referiert

werden. Bei der Spezifikation der Slots kann man ihre Namen und optionale Initialisierungswerte als Lisp-Ausdrücke angeben. Letztere dienen der Slot-Initialisierung, falls beim Aufruf von `make-instance` keine expliziten Initialisierungswerte angegeben werden. Die Auswertung der Initialisierungsausdrücke erfolgt zur Instanzerzeugungszeit in der lexikalischen Umgebung der Klassendefinition[1]. Ist die Liste der direkten Superklassen leer, so wird automatisch die Oberklasse aller Klassen `standard-object` als direkte Superklasse angenommen. Die genaue Syntax von `defclass` ist in Abbildung 6.2 beschrieben.

```
(defclass class-name (slot-spec*) (superclass-name*)
  [ :metaclass metaclass-name ])

slot-spec ::= slot-name | (slot-name lisp-form)
```

Abbildung 6.2: Syntax der Klassen-Definition in MCS 0.5.

Methodendefinition und Methodenkombination

Methoden werden mit dem syntaktischen Konstrukt `defmethod` definiert. Methoden sind hier keine Objekte im Sinne des Objektsystems. Sie sind daher auch auf der Metaobjektebene verborgen. Der einzige Weg, Methoden zu aktivieren, geht über das Versenden von Nachrichten. Methodennamen können im Unterschied zu Klassen-Namen nicht als Bezeichner verwendet werden, weil das Konstrukt `defmethod` keine Bindung erzeugt. Sie dienen nur als Schlüsselwörter in der internen Methodentabelle einer Klasse.

Wie die Syntax der Methodendefinition in Abbildung 6.3 deutlich macht, hat das Empfängerobjekt eine Sonderstellung. Seine Klasse wird in der Methodendefintion als erste genannt, dann erst der Name der Methode und schließlich ihre Parameter. Im Methodenrumpf kann das Empfänger-Objekt über den Sonderbezeichner `self` referiert werden.

MCS 0.5 unterstützt die sogenannte *Standard-Methodenkombination* aus CLOS, die in ihrer Funktionalität der *Demon-Methodenkombination* und sogenannten *Whopper-Methoden* in Flavors entspricht. Dabei kann es mehrere Methoden mit dem gleichen Namen geben, auch für ein und dieselbe Klasse. Die Methoden unterscheiden sich dann aber in ihrer Rolle, die in der Definition über den *qualifier* zwischen dem Klassen- und dem Methodennamen zum Ausdruck gebracht werden kann. Fehlt ein *qualifier*, so handelt es sich um eine *Primärmethode*. Das Ergebnis der Primärmethode stellt in der Regel (ohne *Around-Methoden*) auch das Ergebnis eines Sendeereignisses dar. Mit dem Schlüsselwort `:before` qualifizierte Methoden spezifizieren zusätzliche Aktionen, die vor der Primärmethode auszuführen sind. Mit dem Schlüsselwort `:after` qualifizierte Methoden spezifizieren Aktionen, die nach der Primärmethode auszuführen sind. Typisches Beispiel für *After-Methoden* sind benutzerdefinierte Initialisierungsmethoden (mit dem Namen `:init`). Before- und After-Methoden werden auch als *demons* oder als Zwiebel-Schalen, mit der Primärmethode in der Mitte, bezeichnet. Eine *Around-Methode* umklammert dann eine ganze Zwiebel. Im Methodenrumpf von Primär- und Around-Methoden kann das syntaktische Konstrukt

[1] Faktisch ist dies in CommonLisp die globale Lisp-Umgebung, die in CommonLisp zudem dynamisch ist.

`call-next-method` aufgerufen werden. Findet nun ein Sendeereignis statt, so kommen die anwendbaren Methoden, d. h. die direkten und die ererbten, wie folgt zur Auführung:

- Falls es Around-Methoden gibt, wird die speziellste Around-Methode ausgeführt. Wird in ihr `call-next-method` aufgerufen, so wird die nächst-allgemeinere Around-Methode ausgeführt. Das Ergebnis der speziellsten Around-Methode wird zurückgeliefert.
- Falls es keine Around-Methoden gibt oder falls in der allgemeinsten Around-Methode `call-next-method` aufgerufen wird, kommen folgende Schritte zur Ausführung:
  1. Falls vorhanden, werden alle Before-Methoden ausgeführt, die spezielleren zuerst.
  2. Die (obligatorische) speziellste Primär-Methode wird ausgeführt. Wird in ihr `call-next-method` aufgerufen, so wird die nächst-allgemeinere Primär-Methode ausgeführt. Wird in der allgemeinsten Primär-Methode `call-next-method` aufgerufen, wird ein Fehler gemeldet.
  3. Falls vorhanden, werden alle After-Methoden ausgeführt, die allgemeineren zuerst.
  4. Das Ergebnis der speziellsten Primär-Methode wird zurückgeliefert.

Andere Methodenkombinationen bzw. benutzerdefinierte Methodenkombinationen werden in MCS 0.5 nicht unterstützt. Die eben gelieferte Beschreibung der Standardmethodenkombinationen ist etwas kompliziert, weil das zugrundeliegende Konzept zu komplex ist. In Kapitel 8.3.4 auf Seite 265 stelle ich ein einfacheres und gleichzeitig abstrakteres Konzept der deklarativen Methodenkombination vor.

```
(defmethod (class-name [ qualifier ] method-name) lambda-list
  [ :documentation string ]
  body)

lambda-list ::= ( { parameter }* [ &rest parameter ] )
qualifier ::= :before | :after | :around
body ::= method-form*
method-form ::= lisp-form | next-method-form | slot-access-form | send-self-form
next-method-form ::= (call-next-method { argument }*)
slot-access-form ::= (get-slot slot-name) |
                     (setf (get-slot slot-name) method-form)
send-self-form ::= (send-self method-name { argument }*)
```

Abbildung 6.3: Syntax der Methodendefinition in MCS 0.5.

Aus der Syntax der Methodendefinition (Abbildung 6.3) geht eine weitere Restriktion gegenüber Flavors bzw. CommonLisp hervor: In der *lambda-list* werden geforderte und Rest-Parameter erlaubt, aber keine `&key`-, `&optional`-, `&allow-other-keys` oder `&aux`-Parameter. Aus Kompatibilitätsgründen zu Flavors besitzt das Konstrukt `call-next-method` optionale Parameter. Von ihrer Benutzung wird jedoch abgeraten.

Instanzerzeugung und Initialisierung

Instanzen werden in MCS 0.5 mit dem Konstrukt `make-instance` erzeugt, das aus Effizienzgründen als Makro definiert wurde. Es kann aber auch als Funktion realisiert werden. Als Makro kann es nicht voll funktional verwendet werden, was auf der Objektebene akzeptabel ist. Auf der Metaobjekteben gibt es ohnehin die Möglichkeit, Instanzen voll dynamisch zu erzeugen, auf die ich später zurückkomme. Abbildung 6.4 zeigt die Syntax von `make-instance`:

`(make-instance` *class-expr* { *init-keyword* *lisp-form* }* `)`

class-expr `::=` *class-name* | *quoted-class-name* | *lisp-form*

quoted-class-name `::= (quote` *class-name*`)` | `'`*class-name*

Abbildung 6.4: Syntax der Instanz-Erzeugung in MCS 0.5.

Das Konstrukt `make-instance` erzeugt eine neue Instanz des ersten Arguments und initialisiert jeden Slot entweder gemäß dem expliziten Initialisierungsargument oder gemäß dem Default-Initialisierungsausdruck aus der Klassendefinition, der bei jeder Instanzerzeugung ausgewertet wird, oder mit `nil`, falls es keinen Default-Ausdruck gibt. Anschließend wird dem neuen Objekt die Nachricht `:init` mit allen Initialisierungsargumenten geschickt, so daß benutzerdefinierte Initialisierungsmethoden zum tragen kommen können. Die systemdefinierte Primärmethode der Klasse `standard-object`, die alle Klassen ererben, ist leer und sorgt nur dafür, daß es zu keiner Fehlermeldung kommt, falls es keine benutzerdefinierte Primärmethode gibt. In der Regel definieren Benutzer nur After-Initialisierungsmethoden.

Aus Kompatibilitätsgründen zu Flavors ist die Syntax etwas komplizierter als sie sonst sein müßte. Da Klassenobjekte in MCS 0.5 an ihre Namen gebunden werden, muß ihren Namen in einem Aufruf von `make-instance` auch kein `quote` vorangestellt werden. Man darf es aber, wie in Flavors üblich, tun. Das erste Argument kann auch ein beliebiger Lisp-Ausdruck sein, dessen Auswertung ein Klassenobjekt liefert. Als weitere Argumente können alternierend Schlüsselwörter und Lisp-Ausdrücke spezifiziert werden. Insbesondere können alle Slotnamen beginnend mit einem Doppelpunkt als *init-keyword* verwendet werden, um den jeweiligen Slot mit dem Auswertungsergebnis der entsprechenden *lisp-form* zu initialisieren.

Slotzugriff

In Flavors gibt es zwei Alternativen auf Slots (bzw. Instanzvariablen) zuzugreifen. Der abstraktere Zugriff erfolgt über das Versenden entsprechender Nachrichten. In Methodenrümpfen können Slots des Empfängerobjekts wie normale Lisp-Variablen referiert werden. Dafür muß man aber den Methodenrumpf parsen, analysieren (*code walking*) und entsprechende Substitutionen vornehmen. Will man es richtig machen, so ist dies sehr aufwendig, weil man unzählige Besonderheiten von CommonLisp beachten muß und schließlich

ein halbes Compiler-Oberteil realisieren würde. Das wäre nicht im Sinne der Einbettungstechnik. Daher wird in MCS 0.5 an Stelle der zweiten Alternative, auf Slots zuzugreifen, ein explizites Konstrukt `get-slot` als Makro bereitgestellt, daß nur in Methodenrümpfen verwendet werden darf:

`(get-slot` *slot-name*`)`
`(setf (get-slot` *slot-name*`)` *new-value*`)`

Das Objekt des Zugriffs ist implizit das Empfängerobjekt der Nachricht. Fehlt dem Objekt ein mit *slot-name* benannter Slot, so wird ein Fehler gemeldet.

Zusätzlich gibt es die aus CLOS bekannte Funktion `slot-value`, die auch außerhalb von Methoden verwendet werden darf, um auf Slots zuzugreifen:

`(slot-value` *object slot-name*`)`
`(setf (slot-value` *object slot-name*`)` *new-value*`)`

Diese verhalten sich genau so, wie `get-slot` bzw. `(setf get-slot)` in Methodenrümpfen. In MCS 0.5 gibt es kein spezialisierbares Feldzugriffs-Protokoll, wie in CLOS MOP oder ΤΕΛΟΣ.

Systemdefinierte Klassen und Methoden

Um allen Objekten des Objektsystems ein gemeinsames Mindestverhalten zu geben, wird die systemdefinierte Klasse `standard-object` bereitgestellt. Direkte Instanzen von `standard-object`, die man mit `make-instance` erzeugen kann, haben keine Slots, außer dem speziellen Slot für die Referenz auf die eigene Klasse. Entsprechende Zugriffsmethoden werden auf der Metaobjektebene bereitgestellt. Für `standard-object` sind keine Initialisierungs-Schlüsselwörter definiert. Folgende Primärmethoden legen das Standardverhalten aller Objetke fest:

- `:init` { *argument* }*
  Systemdefinierte Default-Methode, die nichts tut. Sie kann in benutzerdefinierten Klassen spezialisiert werden, in der Regel als After-Methoden.
- `:default-handler` *method-name* { *argument* }*
  Falls bei einem Sendeereignis keine anwendbare Methode gefunden wird, wird `:default-handler` mit ihrem Namen und den ursprünglichen Argumenten aufgerufen. Die systemdefinierte Methode ruft unmittelbar `:error-handler` mit den gleichen Argumenten auf. Sie kann in benutzerdefinierten Klassen spezialisiert werden.
- `:error-handler` *method-name* { *argument* }*
  Systemdefinierte Methode unterbricht die Programmausführung und produziert eine Fehlermeldung. Sie kann in benutzerdefinierten Klassen spezialisiert werden.
- `:describe`
  Systemdefinierte Methode produziert eine Objektbeschreibung mit der Angabe seiner Klasse und seiner Slotwerte.
- `:class-name`
  Liefert den Klassennamen des Objekts. Wird benötigt, um z. B. benutzerspezifische `:describe` Methoden zu implementieren.

- `:operation-handled-p` *method-name*
  Testet, ob eine eigene oder ererbte Methode mit dem Namen *method-name* vorhanden ist. Liefert entsprechend `t` oder `nil` zurück.

- `:send-if-handles` *method-name* { *argument* }*
  Testet, ob eine eigene oder ererbte Methode mit dem Namen *method-name* vorhanden ist. Falls ja, wird diese mit den Argumenten aufgerufen und das entsprechende Ergebnis zurückgeliefert. Sonst ist das Ergebnis `nil`.

Streng genommen können alle eben genannten Methoden, bis auf `:init`, als reflektiv eingestuft werden. Konsequenterweise gehören sie mehr in die Metaobjektebene. Aus Kompatibilitätsgründen mit FLAVORS werden sie hier der Objektebene zugerechnet.

6.2.3 Sprachkonstrukte der Metaobjektebene

Die Sprachkonstrukte der Metaobjektebene müssen Mittel für die Erzeugung, Initialisierung, Introspektion etc. von Metaobjekten bereitstellen. Entsprechend dem syntaktischen Konstrukt `defclass` auf der Objektebene gibt es hier zusätzlich das Konstrukt `defmetaclass`. Es unterscheidet sich von `defclass` nur dadurch, daß als Default-Superklasse die Klasse `standard-class` dient statt `standard-object`.

Reflektive Standardmethoden

Zusätzlich zu den Standardmethoden auf der Objektebene (siehe oben) werden folgende reflektive Methoden der Klasse `standard-object` bereitgestellt:

- `:isit`
  Liefert die Klasse des Objekts.

- `:class-p`
  Liefert `t`, falls das Objekt ein Klassenobjekt ist, sonst `nil`.

- `:metaclass-p`
  Liefert `t`, falls das Objekt ein Metaklassenobjekt ist, sonst `nil`.

Metaobjektklassen und Methoden

Als einzige Metaobjektklasse wird die Klasse `standard-class` als Subklasse von `standard-object` zur Verfügung gestellt. Von `standard-object` erbt `standard-class` den Slot `isit`, die Referenz auf die eigene Klasse, sowie alle o. g. Methoden.

Die Struktur von Klassenobjekten ist durch folgende Slots definiert:

- `isit`, die Referenz auf die eigene Klasse.

- `name`, der Name der Klasse, wird direkt initialisiert.

- `supers`, die Liste der direkten Superklassen, wird direkt initialisiert.

- `cplist`, die Klassenpräzedenzliste.
- `all-slots`, die Liste aller Slot-Namen.
- `all-slot-defaults`, die Liste von zweielementigen Listen aus dem Slotnamen und dem Default-Ausdruck.
- `own-slots`, die Liste der direkten Slotnamen, wird direkt initialisiert.
- `methods`, die Methoden-Hashtabelle.
- `basicnew-fn`, die Konstruktor-Funktion, die von `make-instance` aufgerufen wird, um eine neue Instanz zu erzeugen.
- `slot-accessor-fn`, die *Slotpositionsfunktion*, die zu gegebenem Slotnamen die Slotposition für alle direkten Instanzen dieser Klasse zurückliefert.
- `subclasses`, die Liste der direkten Subklassen.

Auf der Metaobjektebene können Instanzen von `standard-class` mit `make-instance` erzeugt werden. Dabei müssen folgende Initialisierungs-Schlüsselwörter angegeben werden:

- `:name` *class-name*
- `:supers` *direkt-superclasses-list*
- `:own-slots` *direkt-slot-specifications-list*

Die entsprechenden Slots werden direkt bei der Allozierung eines neuen Klassenobjekts mit den angegebenen Werten initialisiert, so daß sie anschließend gelesen werden können. In benutzerdefinierten Initialisierungsmethoden können die entsprechenden Initialisierungsargumente über diese Schlüsselwörter zugegriffen werde, z. B. mit `(getf arguments :name)`.

Klasseninstanzen können nicht nur über die Operation `make-instance`, sondern auch über die Nachricht `:new` an das entsprechende Klassenobjekt erzeugt werden. Dabei gelten die gleichen Initialisierungs-Schlüsselwörter wie für `make-instance`. Die folgenden Methoden von `standard-class` sorgen für die vollständige Initialisierung von Klassenobjekten:

- `:new` { *init-keyword value* }*
  Liefert eine neue Instanz der Klasse. Die systemdefinierte Default-Methode ruft dafür die mit Hilfe von `compute-basicnew-fn` berechnete Konstruktor-Funktion (siehe unten) auf. `:new` kann in benutzerdefinierten Metaklassen spezialisiert werden, in der Regel als After-Methoden.
- `:init` { *init-keyword value* }*
  Systemdefinierte Primär-Methode führt die Initialisierung des Klassenobjekts durch. Dafür ruft sie die unten aufgeführten Methoden auf. Sie kann in benutzerdefinierten Klassen spezialisiert werden, in der Regel als After-Methoden.

- `:compute-cplist`
  Berechnet die Klassenpräzedenzliste nach der Strategie *depth-first-left-to-right-up-to-joins* [Bretthauer *et al.*, 1989c]. Sie kann in benutzerdefinierten Klassen spezialisiert werden.

- `:inherit-slots-with-defaults`
  Berechnet alle Slots einer Klasse, mit ihren Defaults, gemäß der Klassenpräzedenzliste. Doppelte Vorkommen von Slots mit dem gleichen Namen werden entfernt, so daß der speziellste übrigbleibt. Sie kann in benutzerdefinierten Klassen spezialisiert werden.

- `:compute-slot-accessor-fn`
  Berechnet eine *Slotpositionsfunktion*, die zu gegebenem Slotnamen seine Position als Integer-Wert zurückliefert. Sie kann in benutzerdefinierten Klassen spezialisiert werden.

- `:extend-subclasses-of-supers`
  Trägt die Klasse als direkte Subklassen in den direkten Superklassen ein. Benutzer sollten diese Methode nur in After-Methoden spezialisieren.

- `:compute-slot-access-methods`
  Berechnet die Lese- und Schreibmethoden. Sie kann in benutzerdefinierten Klassen spezialisiert werden.

- `:compute-basicnew-fn`
  Berechnet die Konstruktor-Funktion, die von `make-instance` und von der Methode `:new` aufgerufen wird. Die berechnete Konstruktor-Funktion erwartet als geforderten Parameter diese Klasse und als optionale Schlüsselwort-Parameter (`&key`) alle Slotnamen dieser Klasse mit einem Doppelpunkt am Anfang. `:compute-basicnew-fn` kann in benutzerdefinierten Klassen spezialisiert werden.

Um die Spezialisierung dieser Methoden von `standard-class` zu ermöglichen, mußten in der Spezifikation einige Implementierungsdetails offengelegt werden, insbesondere alle Slots sowie die Struktur ihrer Werte. Das liegt auch daran, daß Slotannotationen und Methoden keine Objekte des Objektsystems sind. Um sie zu benutzen und zu erweitern, muß man auf die Implementierungsebene absteigen.

6.2.4 Hilfsmittel der Programmentwicklungsumgebung

Um eine spezielle Unterstützung bei der Verwendung objektorientierter Sprachkonstrukte über die üblichen Lisp-Entwicklungswerkzeuge hinaus zu bieten, werden einige weitere Methoden und Funktionen bereitgestellt. Für alle Objekte, repräsentiert durch die Klasse `standard-object`, sind die Methoden `:describe` und `:which-operations` definiert. Die erste liefert eine Zustandsbeschreibung eines Objekts, zeigt also seine Slots. Die zweite gibt eine Liste der Methodennamen zurück, die für ein Objekt über Nachrichtenereignisse aufgerufen werden können, zeigt also sein Protokoll.

Eine ähnliche Aufgabe haben die Methoden `:get-protocol` und `:get-local-protocol`, die auf allen Klassenobjekten ausführbar sind. Die erste liefert alle Methodennamen, einschließlich der geerbten, während die zweite die für eine Klasse direkt definierten Methodennamen liefert.

Über die eben genannten statischen Programmaspekte hinaus wird das dynamische Verhalten von Klassen und Instanzen durch sogenannte *Trace-Operationen* unterstützt:

`(mcs-trace` *class method-name*`)`
`(mcs-untrace` *class method-name*`)`

Wurde das *tracing* einer Methode der Klasse *class* mit dem Namen *method-name* aktiviert, so werden bei ihrem Aufruf entsprechende Ausgaben generiert, die den internen Ablauf der Aufrufe von Before-, Primär-, After- und Around-Methoden sowie ihrer Ergebnisse transparent machen. Das *Tracing* kann auch über die Methoden `:trace-methods` und `:untrace-methods` ein- bzw. ausgeschaltet werden.

6.3 Implementierung von MCS 0.5

Die Implementierung von MCS 0.5 erfolgte weitgehend nach den Richtlinien aus Kapitel 5. Dabei bilden die Schritte 1 bis 5 aus dem Abschnitt 5.3 das Implementierungsgerüst. Auf Abweichungen werde ich explizit eingehen und sie in der zusammenfassenden Bewertung berücksichtigen.

Da das Bootsrapping des Kerns ohne die Hilfe von syntaktischen Konstrukten erfolgt, können die Schritte 2 (vorläufige Syntax) und 4 (endgültige syntax) zusammengefaßt werden. Dies ist möglich, weil der Objektsystemkern, bestehend aus den zwei Klassen `standard-object` und `standard-class` klein genug ist. Ebenso hält sich die Zirkularität des Kerns noch in Grenzen. Erstens ist die Klasse `standard-class` Instanz von sich selbst. Der zweite Zyklus ist indirekt: `standard-class` ist Subklasse von `standard-object` und `standard-object` ist Instanz von `standard-class`. Bevor die eine Klasse vollständig initialisiert werden kann, muß die andere schon da sein und umgekehrt. Methoden und Slotannotationen sind keine Instanzen von Objektsystem-Klassen. Sie müssen daher nicht in die Bootsrapping-Phase einbezogen werden. Die Implementierung entsprechender Konstrukte kann daher vorgezogen werden.

6.3.1 Schritt 1: Objektrepräsentation und elementare Operationen

Alle Objekte werden in MCS 0.5 als Lisp-Strukturen mit genau einem Struktur-Slot repräsentiert. Mit der folgenden Definition wird ein neuer Strukturtyp mit dem Namen `mcsobject` definiert:

```
(defstruct (mcsobject (:conc-name mcs-)
                      (:print-function print-mcs))
   env)
```

Der einzige Struktur-Slot `env` von `mcsobject` ist ein Platzhalter für einen Vektor mit den zu repräsentierenden Slot-Werten von Objekten des Objektsystems. Die Referenz eines jeden Objekts auf seine Klasse wird dabei ebenfalls als Slot-Wert des Objektsystems realisiert. An dieser Stelle wird die Uniformität und Reflektivität auf die Spitze getrieben. Dies darf aber nicht darüber hinwegtäuschen, daß die Referenz auf die eigene Klasse eine Sonderstellung einnimmt. Der Zugriff auf die eigene Klasse kann eben nicht genauso, wie auf alle anderen Slot-Werte über Nachrichtenversenden erfolgen. Um eine Nachricht zu verarbeiten, benötigt ein Objekt bereits die eigene Klasse. Eine uniforme Implementierung würde an dieser Stelle eine Endlosschleife produzieren. Um sie abzufangen und aufzubrechen, sind besondere Vorkehrungen erforderlich. Daher wird in MCS 1.0 die spezielle Stellung der Klassen-Referenz in einem Objekt gleich in der Strukturtyp-Definition berücksichtigt, wie schon in Kapitel 5 beschrieben wurde.

[Bild wie Abb. 5.6]

Die spezifizierte Option `conc-name` in der obigen Defintion hat nur eine optische Funktion. Sie bewirkt, daß die automatisch generierten Lese- und Schreiboperationen mit dem Präfix `mcs-` beginnen und nicht mit `mcsobject`. Wichtiger ist die Option `:print-function`, um die externe Repräsentation eines Objekts zu kontrollieren, was wegen der inherent zirkulären Struktur von `standard-class` notwendig ist.

Mit der Strukturtyp-Definition stehen nun folgende Operationen zur Verfügung:

`(make-mcsobject :env` *slot-value-vector*`)`
`(mcsobject-p` *object*`)`
`(mcs-env` *object*`)`
`(setf (mcs-env` *object*`)` *slot-value-vector*`)`

Nun wird festgelegt, wie der Vektor der Slotwerte aufgebaut werden soll. An erster Position steht in jedem Objekt seine eigene Klasse. Also wird ein Makro `index-of-isit` definiert, das als Ergebnis der Makroexpansion den konstanten Wert 0 liefert[2]. Der primitive Zugriff auf die Klasse eines Objekts sieht dann folgendermaßen aus:

`(svref (mcs-env` *object*`) (index-of-isit))`
`(setf (svref (mcs-env` *object*`) (index-of-isit))` *class*`)`

Um generell auf einen Slot mit bekannter Position zuzugreifen wird das Implementierungsprimitiv `mcs-slot-value` als Makro eingeführt:

```
(defmacro mcs-slot-value (object slot-position)
  `(svref (mcs-env ,object) ,slot-position))
```

Dadurch können möglich Änderungen der Objektrepräsentation mit weniger Quellcode-Modifikationen umgesetzt werden. Für `(setf mcs-slot-value)` ist weiter nichts erforderlich, weil es in Lisp für den Operator des Makroexpansionsergebnisses `svref` eine entsprechende `setf`-Definition schon gibt.

[2]Statt einem Makro kann man hier auch eine Konstante definieren. Mit einem Makro behält man sich größere Implementierungsfreiheiten offen. Beispielsweise kann ein Makro in unterschiedlichen Kontexten verschiedene Expansionsergebnisse liefern.

Implementierung von Klassen

Bei der Implementierung von Klassen werden ebenso feste Positionen der systemdefinierten Slots in Klassenobjekten vorgegeben. Ohne diese Annahme fester Positionen und entsprechender primitiver Zugriffe würde man wieder mit Endlosschleifen konfrontiert. Um z. B. auf einen Slot eines Objekts zuzugreifen, muß die Slotposition über die Klasse des Objekts bestimmt werden, was seinerseits einen Slotzugriff auf das Klassenobjekt erfordert usw. Die klassenspezifische *Slotpositionsfunktion*, die zu gegebenem Slotnamen seine Position zurückliefert, muß also auf primitive Weise im Klassenobjekt gefunden werden können. Ihr konstanter Wert wird mit dem Makro `index-of-slot-accessor-fn` festgelegt:

```
(svref (mcs-env object) (index-of-slot-accessor-fn))
```

Aus Effizienzgründen werden auch die anderen Slots von Klassenobjekten mit festen Positionen versehen, ohne daß es konzeptionell, wie bei den Slots `isit` und `slot-accessor-fn`, erforderlich wäre. Die Annahme fester Positionen und die Verwendung eines primitiven Zugriffs schränkt natürlich die möglichen Spezialisierungen der Klasse `standard-class` bzgl. der Repräsentation von Klassenobjekten grundsätzlich ein. Sie dürfen nur erweiternd sein und sie dürfen an den Positionen der Systemslots nicht rütteln.

Implementierung von Methoden

Methoden werden nicht als Objekte des Objektsystems realisiert, sondern als anononyme Funktionen. Jede Klasse besitzt eine Methoden-Hashtabelle, in der einem *Methodennamen* ein *Methodeneintrag* zugeordnet wird. Der Methodeneintrag muß den Methodenkombinationstyp, die definierten Methoden und ggf. die kombinierte Methode (nur für direkte Instanzen einer Klasse) aufnehmen. Der Methodeneintrag wird als ein weiterer Strukturtyp definiert:

```
(defstruct method-entry
  type
  method-list
  combined-method)
```

Auch wenn nur ein einziger Methodenkombinationstyp realisiert wurde, sieht die Implementierung mit dem Struktur-Slot `type` eine mögliche Erweiterung vor. Die Assoziationsliste `method-list` enthält maximal vier Elemente, je nachdem ob eine Before-, eine Primär-, eine After- oder eine Around-Methode für die gegebene Klasse definiert wurde.

Bei der Implementierung von Methoden muß dafür gesorgt werden,

- daß man in Methodenrümpfen effizient auf die Slots des Empfängerobjekts zugreifen kann,
- daß man das Empfängerobjekt über den speziellen Bezeichner `self` referieren kann und
- daß man den Aufruf der nächst-allgemeineren Methode in Primär- bzw. in Around-Methoden über `call-next-method` ermöglicht.

Die anonymen Funktionen, die Methoden repräsentieren, erhalten daher, neben den benutzerdefinierten, zusätzliche Parameter:

`(lambda (self &class-env &inst-env &caller &next-methods &args` $\{arg\}^*$`)`
`...)`

Da in Before- und After-Methoden kein `call-next-method` erlaubt ist, kann dort auf die Parameter `&caller`, `&next-methods` und `&args` verzichtet werden:

`(lambda (self &class-env &inst-env` $\{\ arg\ \}^*$`) ...)`

Die Semantik des speziellen Bezeichners `self` gleicht nun der eines regulären Parameters. Dies ist auf jeden Fall effizienter als z. B. die Verwendung dynamischer Bindungen. Bei der Implementierung des Nachrichtenversendens müssen die Aufrufe der Methodenfunktionen entsprechend organisiert werden.

Implementierung von Call-Next-Method

Das syntaktische Konstrukt `call-next-method` kann ebenfalls als Makro realisiert werden:

```
(defmacro call-next-method (&rest supplied-args)
  (if supplied-args
    ;; apply next method on supplied arguments
    `(apply-next self &class-env &inst-env &caller &next-methods
                 ,@supplied-args)
    ;; call next method on original arguments, collected in &args
    `(call-next self &class-env &inst-env &caller &next-methods &args)))
```

Dabei werden die Bezeichner `&class-env`, `&inst-env`, `&caller`, `&next-methods` und `&args` nicht-hygienisch verwendet. Obwohl das Sprachkonstrukt `call-next-method` mit expliziten Argumenten, die von den ursprünglichen Argumenten des Sendeereignisses verschieden sind, in Kapitel 4 als problematisch eingeschätzt wurde, ist es hier aus Kompatibilitätsgründen zu Flavors bzw. CLOS nicht eingeschränkt worden. Aus Effizienzgründen gibt es zwei Funktionen, auf die `call-next-method` abgebildet wird: `call-next` und `apply-next`. Sie unterscheiden sich nur darin, daß `apply-next` einen `&rest`-Parameter hat.

```
(defun call-next (self class-env inst-env caller next-methods args)
  (declare (optimize (speed 3) (safety 0)))
  (if (eq caller :primary-caller)
    ;; calling next method from a :primary method
    ;; all next-methods are primary methods
    (let ((next-method (first next-methods)))
      (if next-method
        ;; there is a next :primary method
        (apply next-method
               self class-env inst-env
               :primary-caller (rest next-methods) args
```

```
              args)
        ;; there is no next :primary method
        (error "Can't call next method from primary method.")))
    ;; calling next method from an :around method:
    (let ((around-methods (around-of next-methods)))
      (if around-methods
        ;; there is a next :around method
        (apply (first around-methods)
               self class-env inst-env
               :around-caller (cons (rest around-methods)
                                    (demons-of next-methods))
               args args)
        ;; there is no next :around method
        ;; call :before, :primary and :after methods
        (demon-method-combination self class-env inst-env
                                  :dummy-selector
                                  next-methods
                                  args)))))

(defun apply-next (self class-env inst-env caller next-methods &rest args)
   ...)
```

Implementierung der Slotzugriffe innerhalb von Methoden

Die Konstrukte `get-slot` und `(setf get-slot)`, die nur in Methoden vorkommen dürfen, können nun wie folgt implementiert werden:

```
(defmacro get-slot (slot-name)
  `(svref &inst-env                    ; secret parameter
          ;; get the slot position calling the slot access funktion
          ;; stored in the class of the receiver object
          (funcall (svref &class-env  ; secret parameter
                          (index-of-slot-accessor-fn))
                   ,slot-name)))
```

Die Slotpositionsfunktion wird selbst über einen festen Index aus dem Klassen-Slotvektor geholt und mit dem gesuchten Slotnamen als Argument aufgerufen. Als Ergebnis liefert sie den Index des gesuchten Slots im Slotvektor der Instanz. Die Slotpositionsfunktion hat beispielsweise für die Klasse `standard-object` folgendes Muster:

```
#'(lambda (slot-name)
    (case slot-name
      (isit (index-of-isit))    ; expands to a constant literal 0
      (t (error "No slot named ~S" slot-name))))
```

Für `(setf get-slot)` ist weiter nichts erforderlich, weil es in Lisp für den Operator des Makroexpansionsergebnisses `svref` eine entsprechende `setf`-Definition schon gibt. Hier

wird gezielt nicht-hygienische Makroexpansion verwendet. Die im Makroexpansionsergebnis vorkommenden Bezeichner bekommen ihre Bedeutung aus dem Kontext des Aufrufs von `get-slot`, nicht der Definition des Makros `get-slot`. Es ist eine gewollte Interaktion mit der Umgebung des Aufrufs. Bei hygienischer Makroexpansion soll eine ungewollte Interaktion verhindert werden. Bietet eine Sprache nur hygienische Makros an, so wird leider auch eine gewünschte Interaktion des Makroexpansionsergebnisses mit seiner Umgebung unterbunden.

Implementierung einer generellen Slotzugriffsfunktion

Die Implementierung der Zugriffsoperation `slot-value` verwendet wie schon `get-slot` die klassenspezifische Slotpositionsfunktion, um die Position des gesuchten Slots zu bestimmen. Anschließend kann ein Indexzugriff auf den Slotvektor innerhalb der Struktur vom Typ `mcsobject` erfolgen:

```
(defun slot-value (object slot)
  (declare (optimize (speed 3) (safety 0)))
  (let ((object-env (mcs-env object)))
    (svref object-env
           (funcall (svref (mcs-env (svref object-env
                                            (index-of-isit)))
                           (index-of-slot-accessor-fn))
                    slot))))

(defun set-slot-value (object slot value)
  (declare (optimize (speed 3) (safety 0)))
  (let ((object-env (mcs-env object)))
    (setf (svref object-env
                 (funcall (svref (mcs-env (svref object-env
                                                  (index-of-isit)))
                                 (index-of-slot-accessor-fn))
                          slot))
           value)))

(defsetf slot-value set-slot-value)
```

Um auch `slot-value` in uniformer Weise mit `setf` benutzen zu können, wird zunächst eine Schreibfunktion `set-slot-value` definiert und mit Hilfe von `defsetf` installiert. Anschließend darf man schreibende Zugriffe mit `(setf (slot-value ...) ...)` notieren.

Implementierung von Zugriffsmethoden

Während der Initialisierung von Klassenobjekten, wird für alle Slots, auch für die ererbten, eine Lese- und eine Schreibmethode generiert. Dieser Initialisierungsteil wird von der Methoden `compute-accessor-methods` erledigt. Da sie nach der Vererbung von Slots aufgerufen wird, stehen alle Slotpositionen für direkte Instanzen der zu initialisierenden

Klasse fest. Entsprechend kann der Zugriff innerhalb der generierten Methoden über einen festen Index erfolgen. Die eigentlichen Funktionen, als Bestandteile der Methodeneinträge, müssen nicht immer wieder neu berechnet werden. Vielmehr kann dieselbe Funktion für die gleiche Slotposition in allen Klassen verwendet werden. Das Objektsystem definiert statisch solche Funktionen für die ersten 64 Positionen, siehe Kapitel 5. Erst wenn vom Benutzer Klassen mit noch mehr Slots definiert werden, müssen neue Funktionen zur Klasseninitialisierungszeit erzeugt werden, die dann aber für weitere Fälle wiederverwendet werden.

Implementierung des Nachrichtenversendens

Mit den getroffenen Implementierungsentscheidungen besteht die Aufgabe bei der Implementierung von `send-message` im wesentlichen darin, möglichst schnell zu prüfen, ob es sich um ein Objekt des Objektsystems handelt. Falls ja, wird der allgemeine Nachrichten-Handler aufgerufen. Sonst wird ein Fehler gemeldet:

```
(defun send-message (self selector &rest args)
  (declare (optimize (speed 3) (safety 1))
           (inline mcsobject-p mcs-env standard-message-handler))
  (if (mcsobject-p self)
    (let* ((inst-env (mcs-env self))
           (class-env (mcs-env (svref inst-env (index-of-isit)))))
      (declare (optimize (speed 3) (safety 0)))
      (standard-message-handler self class-env inst-env selector args))
    (format nil "ERROR in SEND: SEND can't be applied on ~S" self)))
```

Im Konstrukt `send-fast` kann auf den zeitaufwendigen Typtest verzichtet werden:

```
(defun send-fast (self selector &rest args)
  (declare (optimize (speed 3) (safety 0))
           (inline mcs-env standard-message-handler))
  (let* ((inst-env (mcs-env self))
         (class-env (mcs-env (svref inst-env (index-of-isit)))))
    (standard-message-handler self class-env inst-env selector args)))
```

Hier muß das Objekt nur dereferenziert werden bzw. seine äußere Schale für schnellere Slotzugriffe innerhalb von Methoden abgestreift werden. Bei `send-self` entfällt schließlich auch diese Aktion, weil innerhalb von Methoden direkt auf die Slotvektoren der Instanz (`&inst-env`) und der Klasse (`&class-env`) zugegriffen werden kann:

```
(defmacro send-self (selector &rest args)
  `(standard-message-handler self
                             &class-env
                             &inst-env
                             ,selector
                             (list ,@args)))
```

Man muß sich natürlich fragen, ob nicht die Einführung von so vielen zusätzlichen Parametern zu Effizienzverlusten führt. Im allgemeinen sind Funktionsaufrufe mit wenigen Parametern schneller. Aus praktischen Erfahrungen[3] wurde die Entscheidung hier zugunsten schnellerer Zugriffe innerhalb von Methoden getroffen. In MCS 1.0 wird diese Entscheidung teilweise revidiert, was aber auch mit dem Umstieg auf generische Funktionen mit Multimethoden zusammenhängt.

6.3.2 Schritt 2 und 4: syntaktische Konstrukte

Der erreichte Implementierungsstand erlaubt nun, die wichtigsten syntaktischen Konstrukte von MCS 0.5 `defclass`, `defmethod` und `make-instance` zu realisieren. Die Beschränkung der Reflektion auf Klassenobjekte vereinfacht das Bootstrapping, so daß man auf vorläufige syntaktische Konstrukte ganz verzichten kann. Somit können die Schritte 2 und 4 zusammengefaßt werden.

Instanzerzeugung

Um Objekte effizient erzeugen zu können, wird für jede Klasse eine Konstruktor-Funktion berechnet. Diese hat z. B. für die Klasse `standard-object` folgendes Muster:

```
#'(lambda (class)
    (send-fast (make-mcsobject :env (vector class))
               :init))
```

Es wird ein neues Lisp-Objekt des Strukturtyps `mcsobject` erzeugt, dessen einziger Struktur-Slot `env` mit einem neuen Vektor der Länge 1 initialisiert wird. Das einzige Element dieses Vektors erhält die Klasse `standard-object` als Wert. Das Klassenobjekt wird als Argument der Konstruktorfunktion übergeben. Abschließend wird dem neuen Objekt die Nachricht `:init` geschickt. Da an dieser Stelle der Typ des Objekts bekannt ist, kann auf eine Typprüfung verzichtet und das schnellere Konstrukt `send-fast` verwendet werden.

Die Aufgabe von `make-instance` besteht dann nur darin, diese Konstruktor-Funktion aufzurufen:

```
(defmacro make-instance (a_class &rest initializations)
  `(let ((class ,(if (and (listp a_class) (eq (first a_class) 'quote))
                     (second a_class)
                     a_class)))
     (funcall (mcs-slot-value class (index-of-basicnew-fn))
              class ,@initializations)))
```

Dabei wird der Sonderfall behandelt, daß das erste Argument ein `quote`-Ausdruck ist. Da `make-instance` ein Makro ist, wird dieser syntaktische Ballast zur Makroexpansions- und somit zur Übersetzungszeit abgestreift. Zur Ausführungszeit braucht keine Quellcodeanalyse oder -interpretation erfolgen.

[3] Zumindest in Macintosh Common Lisp. Darin können sich Lisp-Systeme unterscheiden.

Klassen-Definition

Da Klassen Objekte erster Ordnung sind, kann auch das Konstrukt `defclass` einfach auf `make-instance` abgebildet werden.

```
(defmacro defclass (class-name instance-variables superclasses
                    &key (metaclass 'standard-class))
  `(setq ,class-name
         (make-instance ,metaclass
                        :name ',class-name
                        :supers ,(if (null superclasses)
                                     `(list standard-object)
                                     `(list ,@superclasses))
                        :own-slots ',instance-variables)))
```

Alle notwendigen Berechnungen, die im Zusammenhang mit der Klassendefinition stehen, finden innerhalb der Initialisierungsmethode `:init` statt. Natürlich darf `defclass` erst dann verwendet werden, wenn die Klassen `standard-class` und `standard-object` bereits existieren. Dies ist erst nach Abschluß der Bootstrapping-Phase gegeben.

Methodendefinition

Da Methoden keine Objekte des Objektsystems darstellen, werden sie mit reinen Lisp-Mitteln realisiert. Zunächst findet zur Makroexpansionszeit eine einfache syntaktische Vorverarbeitung statt. Das Ergebnis der Makroexpansion ist dann ein `let`-Ausdruck, der zur Ladezeit des Programms ausgeführt wird. Dann muß geprüft werden, ob es einen entsprechenden Methodeneintrag schon gibt. Falls ja, wird die neue Methode hinzugefügt. Sie kann dabei auch eine alte Methode ersetzen. Falls nicht, wird ein neuer Methodeneintrag erzeugt und der Methoden-Hashtabelle hinzugefügt. Auf jeden Fall muß geprüft werden, ob kombinierte Methoden im Cache durch die neue Definition ungültig werden. Der Methoden-Cache muß also aktualisiert werden.

```
(defmacro defmethod ((a_class . qualifier-and-selector) parameters
                     &rest body)
  (let ((qualifier (...))   ; parse qualifier-and-selector
        (selector (...)))  ; parse qualifier-and-selector
    ;; the result of macroexpansion
    `(let ((method-entry    ; look for a method entry
            (gethash ,selector (mcs-slot-value ,a_class
                                               (index-of-methods))))
           (new-method-fn   ; create a new method function
            (function ,(make-lambda-expr qualifier parameters body))))
       (if method-entry
         ;; update the old entry
         (add-qualified-method ,qualifier method-entry new-method-fn)
         ;; create a new method entry and put it into the method table
         (setf (gethash ,selector (mcs-slot-value ,a_class
```

```
                                                    (index-of-methods)))
            (make-method-entry :type 'standard
                               :methods-list
                               (acons ,qualifier new-method-fn ())
                               :combined-method nil) ))
      ;; there might be invalid combined and cached methods
      (remove-invalid-combined-methods ,a_class ,selector)
      ',selector)))
```

Wichtig ist hier zu erwähnen, daß im Expansionsergebnis von `defmethod` ein auszuwertender Ausdruck der Form `(function (lambda (...) ...))`[4] steht. Dieser kann somit vom Datei-Compiler übersetzt und optimiert werden. Zur Lade- bzw. Ausführungszeit wird an dieser Stelle eine kompilierte Funktion erzeugt, die effizient ausgeführt werden kann. Auch ein Interpretierer, der Funktionen vor ihrer Ausführung inkrementell übersetzt, kann dann Methoden in gleicher Weise behandeln. Dies ist ein Kernpunkt einer effizienten Implementierung des Objektsystems mit Hilfe der Einbettungstechnik.

6.3.3 Schritt 3: Bootstrapping des Kerns

Wie schon oben ausgeführt, müssen die Metaobjekte des Objektsystem-Kerns manuell erzeugt und initialisiert werden, bevor die Implementierung der objektorientierten Sprachkonstrukte funktionieren kann. Da es in MCS 0.5 nur zwei Metaobjekte gibt, können sie hier vollständig im Quellcode beschrieben werden. Zunächst werden die Objekte `standard-class` und `standard-object` erzeugt, und zwar, soweit wie möglich, schon mit den richtigen Slotwerten. Die Slots `isit`, `supers` und `cplist` von `standard-class` sowie die Slots `supers` und `cplist` von `standard-object` können erst richtig gesetzt werden, wenn beide Objekte bereits existieren.

```
(setq standard-class
      (make-mcsobject
       :env
       (vector
        'isit           ; :isit will be set below
        'standard-class ; :name
        nil             ; :supers will be set below
        nil             ; :cplist will be set below
        '(isit          ; :all-slots
          name supers cplist all-slots all-slot-defaults own-slots
          methods basicnew-fn slot-accessor-fn subclasses)
        '((name nil)     ; :all-slot-defaults
          (supers nil)(cplist nil)(all-slots nil)(all-slot-defaults nil)
          (own-slots nil)(methods (make-hash-table :test #'eq))
          (basicnew-fn nil)(slot-accessor-fn nil) (subclasses nil))
        '(name           ; :own-slots
          supers cplist all-slots all-slot-defaults own-slots
```

[4] Bzw. `#'(lambda (...) ...)` als Kurzschreibweise

```
          methods basicnew-fn slot-accessor-fn subclasses)
        (make-hash-table :test #'eq)   ; :method table
        ;; :basicnew-fn
        #'(lambda (isit &key (name nil) (supers nil) (own-slots nil))
            (declare (optimize (speed 3) (safety 0)))
            (send-fast
              (make-mcsobject
                :env
                (vector isit name supers nil nil nil own-slots
                        (make-hash-table :test #'eq)
                        nil nil nil))
                :init :name name :supers supers :own-slots own-slots))
        ;; :slot-accessor-fn
        #'(lambda (slot-name)
              (declare (optimize (speed 3) (safety 0)))
              (case slot-name
                (isit              (index-of-isit))
                (name              (index-of-name))
                (supers            (index-of-supers))
                (cplist            (index-of-cplist))
                (all-slots         (index-of-all-slots))
                (own-slots         (index-of-own-slots))
                (all-slot-defaults (index-of-all-slot-defaults))
                (methods           (index-of-methods))
                (basicnew-fn       (index-of-basicnew-fn))
                (slot-accessor-fn  (index-of-slot-accessor-fn))
                (subclasses        (index-of-subclasses))
                (t (error "No slot named ~S." slot-name))))
         ;; :subclasses
         nil)))

;;; Slot 'isit of standard-class has to be set to itself
(setf (svref (mcs-env standard-class) (index-of-isit)) standard-class)

;;;  Hand coded object standard-object

(setq standard-object
      (make-mcsobject
       :env
        (vector standard-class    ; :isit
                'standard-object  ; :name
                nil               ; :supers will be set below
                nil               ; :cplist will be set below
                '(isit)           ; :all-slots
                nil               ; :all-slot-defaults
                '(isit)           ; :own-slots
```

```
                (make-hash-table :test #'eq) ; :methods
                #'(lambda (isit)  ; :basicnew-fn
                    (send-fast (make-mcsobject :env (vector isit))
                               :init))
                #'(lambda (slot)   ; :slot-accessor-fn
                    (case slot
                      (isit  (index-of-isit))
                      (t (error "no slot"))))
                ; :subclasses
                (list standard-class))))

;;; Setup mutual recursive structure of the kernel
(setf (slot-value standard-object 'cplist) (list standard-object))
(setf (slot-value standard-class 'supers) (list standard-object))
(setf (slot-value standard-class 'cplist) (list standard-class standard-
object))
```

Nach Abschluß des Bootstrappings können nun die Methoden des Metaobjektprotokolls mit Hilfe des Objektsystems selbst definiert werden, d. h. unter Verwendung des Konstrukts `defmethod`, etc.

6.3.4 Schritt 5: Vervollständigung des Metaobjektprotokolls

Die drei wichtigsten Methoden des Metaobjektprotokolls können hier vollständig im Quellcode wiedergegeben werden. Es sind die Methoden `:init` von `standard-object` sowie die Methoden `:new` und `:init` von `standard-class`:

```
(defmethod (standard-object :init) (&rest inits)
  (declare (ignore inits) (optimize (speed 3) (safety 0)))
  self)

(defmethod (standard-class :new) (&rest inits)
  (declare (optimize (speed 3) (safety 0)))
  (apply (svref &inst-env (index-of-basicnew-fn))
         self inits))

(defmethod (standard-class :init) (&rest inits)
  (declare (optimize (speed 3) (safety 0)) (ignore inits))
  (send-self :compute-cplist)
  (send-self :inherit-slots-with-defaults)
  (send-self :extend-subclasses-of-supers)
  (send-self :compute-slot-accessor-fn)
  (send-self :compute-slot-access-methods)
  (send-self :compute-basicnew-fn)
  self)
```

Natürlich müssen nun auch die in `:init` für `standard-class` aufgerufenen Methoden `:compute-cplist`, `:inherit-slots-with-defaults`, `:extend-subclasses-of-supers`, `:compute-slot-accessor-fn`, `:compute-slot-access-methods` und `:compute-basicnew-fn` gemäß der Spezifikation in Abschnitt 6.2.3 implementiert werden. Die Implementierungsdetails können im Original-Quellcode gefunden werden (siehe http://nathan.gmd.de/persons/harry.bretthauer [todo:?]).

6.3.5 Realisierte Optimierungen

An dieser Stelle sollen die teilweise schon erwähnten Optimierungen nochmal zusammengefaßt werden. Dazu gehören das *dynamische Caching von Methoden*, optimierte *Konstruktoren*, optimierte *Slotzugriffsmethoden* und die speziell für jede Klasse berechnete *Slotpositionsfunktion*.

Das dynamische Caching von Methoden mit einfachem Dispatch entspricht der Lösung aus Kapitel 5.5.1. Zusätzlich muß hier die Methodenkombination berücksichtigt werde. Die Optimierung der Methodenkombination besteht darin, daß man spezielle *Methodenkombinations-Funktionen* definiert, die auf die jeweilige Situation zugeschnitten sind. Sind nur Primärmethoden anwendbar, so kommt die Funktion `simple-method-combination` zum tragen. Gibt es anwendbare Before- oder After-Methoden, wird die Kombinationsfunktion `demon-method-combination` ausgeführt. Erst bei anwendbaren Around-Methoden muß der kompliziertesteste Fall mit `standard-method-combination` berücksichtigt werden.

Die Objekterzeugung wird durch die Berechnung spezieller Konstruktoren für jede Klasse optimiert, wie in Abschnitt 6.3.2 beschrieben. Die Zuordnung von optionalen Schlüsselwort-Parameter zu entsprechenden Slots zur Laufzeit wird dabei auf den eingebauten CommonLisp-Mechanismus der Schlüsselwortparametern regulärer Funktionen abgebildet. Dadurch wird zwar die dynamische Zuordnung nicht vermieden, aber sie wird auf die effizienteste Weise erledigt. Innerhalb der Konstruktorfunktion brauchen keine expliziten Iterationen mehr durchgeführt zu werden.

Um die Slotzugriffe zu optimieren werden zwei Techniken angewandt. Wie in Abschnitt 6.3.1 beschrieben, wird zum einen für jeden Slot in jeder Klasse eine Lese- und eine Schreibmethode generiert. Das Lösungsprinzip hierfür wurde in Kapitel 5.4.1 bereits dargestellt. Zum anderen wird für Zugriffe über `slot-value` und `get-slot` in jeder Klasse eine Slotpositionsfunktion berechnet, die zu gegebenem Slotnamen seine Position im Slotvektor als Index zurückliefert. Diese Funktion besteht im wesentlichen aus einer `case`-Anweisung, so daß entsprechende Compiler-Optimierungen zum tragen kommen können. Auf jeden Fall ist diese Lösung schneller, als die lineare Suche in einer Liste oder einem Vektor. Eine Hashtabelle würde sich erst bei mehr als etwa 30 Slots lohnen[5].

6.4 Performanzmessungen

In diesem Abschnitt werden die Ergebnisse von Performanzmessungen präsentiert und diskutiert. Dabei beginne ich mit einem Vergleichstest, den Jürgen Walther mit den Objektsy-

[5] Dieser Wert hängt natürlich von der jeweiligen Lisp-Implementierung ab und soll nur als Richtgröße dienen.

stemen WFS, MCF, PCL und MCS 0.5 durchgeführt hat [Walther, 1988]. Anschließend stelle ich die Ergebnisse eines eigenen Vergleichstests auf verschiedenen Rechnerplattformen vor.

6.4.1 Erster Vergleichstest

Der Vergleichstest aus [Walther, 1988] wurde mit dem Ziel durchgeführt, das am besten geeignete Objektsystem für BABYLON auszuwählen. Die Tests wurden auf einem Apple Macintosh II in Allegro Common Lisp 1.2.2 mit 5 MB zugeteiltem Arbeitsspeicher durchgeführt. Dabei wurden Sprachkonstrukte von Objektsystemen exemplarisch getestet, ohne BABYLON als Gesamtsystem zu laden. Letzteres wird im nächsten Abschnitt dargestellt.

Wie bereits in Kapitel 5 gesagt, gehören zu den performanzkritischen Aspekten eines Objektsystems die Erzeugung und Initialisierung von Objekten, die Slotzugriffsoperationen und der generische Dispatch bzw. das Nachrichtenversenden. In [Walther, 1988] wurden folgende Laufzeit-Tests durchgeführt:

- **ft1** erzeugt zur Laufzeit 15 in einem binären Baum angeordnete Klassen.
- **ft2** erzeugt zur Laufzeit 8 Instanzen von Klassen, je eine pro Blatt des Baumes.
- **ft3** greift 100 mal auf alle Slots aller Instanzen über Nachrichtenversenden zu.
- **ft4** setzt 100 mal alle Slots aller Instanzen auf Zufallswerte. Bei WFS und MCF wird dazu Nchrichtenversenden benutzt, bei PCL und MCS 0.5 die Konstruktion `(setf (slot-value instance slot-name) new-value)`.
- **ft5** sendet 100 mal jeder Instanz die Nachrichten `:operation-handled-p` und `:send-if-handles`.
- **ft6** führt ein `compile-flavor-methods` für alle Klassen durch, von denen Instanzen gebildet werden.
- **ft3c** wiederholt **ft3** nach der Kompilierung der Klassen.
- **ft4c** wiederholt **ft4** nach der Kompilierung der Klassen.
- **ft5c** wiederholt **ft5** nach der Kompilierung der Klassen.

Die vollständigen Ergebnisse, die ich aus [Walther, 1988] entnommen habe, sind im Anhang B.1 zu finden. Abbildung 6.5 und Abbildung 6.6 zeigen die relevantesten Testergebnisse. Dabei ist zu beachten, daß alle Werte von **ft2** mit 10 multipliziert wurden, um eine realistischere Proportion zu den anderen Tests zu erreichen. Eigentlich wäre ein Faktor 100 angemessener, was aber die Darstellung als Balkendiagramm erschweren würde[6].

Bei PCL und zunächst auch bei MCS 0.5 gab es kein explizites Konstrukt `compile-flavor-methods`. Vielmehr wird ein ähnlicher Effekt durch das dynamische Caching kombinierter Methoden erreicht.

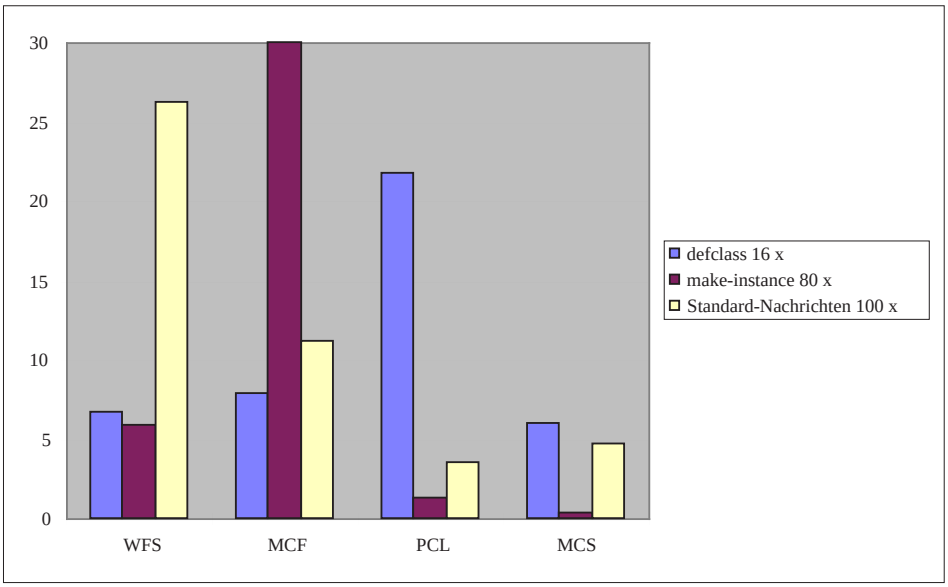


Abbildung 6.5: 1. Vergleich der Ausführungszeiten auf Apple Macintosh II in Allegro Common Lisp 1.2.1 in Sekunden, Teil I.

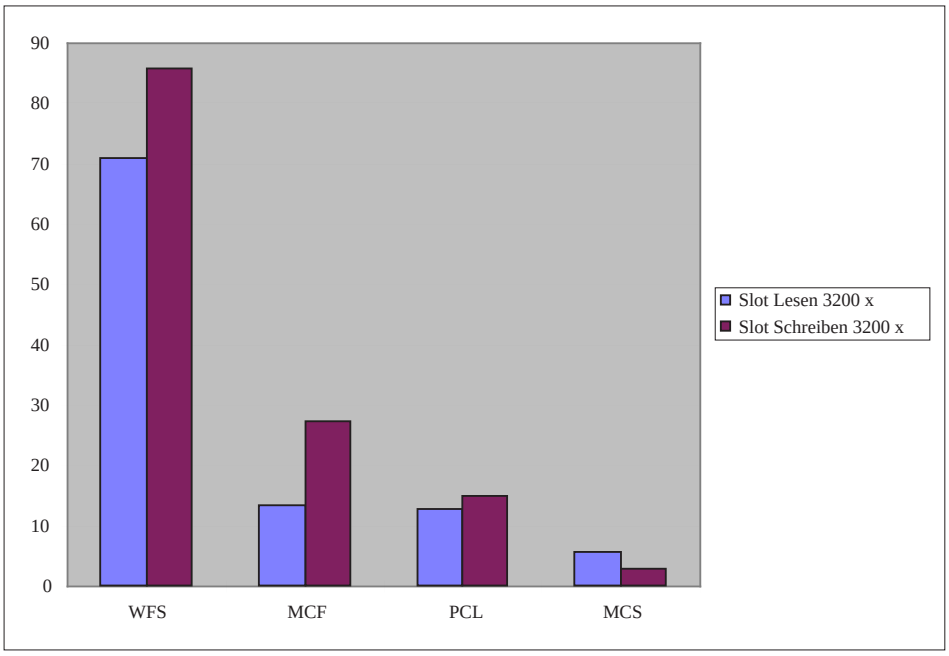


Abbildung 6.6: 1. Vergleich der Ausführungszeiten auf Apple Macintosh II in Allegro Common Lisp 1.2.1 in Sekunden, Teil II

Es mag überraschen, daß die Schreibzugriffe bei MCS 0.5 weniger Zeit benötigen als die Lesezugriffe. Dies ist dadurch zu erklären, daß bei Test **ft3** bzw. **ft3c** WFS und MCF Nachrichtenversenden benutzen, während PCL und MCS 0.5 die Konstruktion `(setf (slot-value instance slot-name) new-value)` verwenden. Wie der zweite Vergleichstest zeigen wird, gleichen die Ergebnisse für Schreibzugriffe über Nachrichtenversenden in MCS 0.5 nahezu denen für Lesezugriffe.

Ebenso wurde der Speicherplatzbedarf verglichen. Dabei fiel vor allem PCL nicht nur durch den Platztbedarf für die Implementierung des Objektsystems, sondern auch durch den Platzbedarf seiner Anwendungen aus dem Rahmen.

> "Allein der Regelprozessor von BABYLON unter PCL braucht mehr Platz als alle Prozessoren von BABYLON unter MCS." [Walther, 1988, S. 3]

Die Ergebnisse des Speicherplatzvergleichs sind in Abbildung 6.7 graphisch dargestellt.

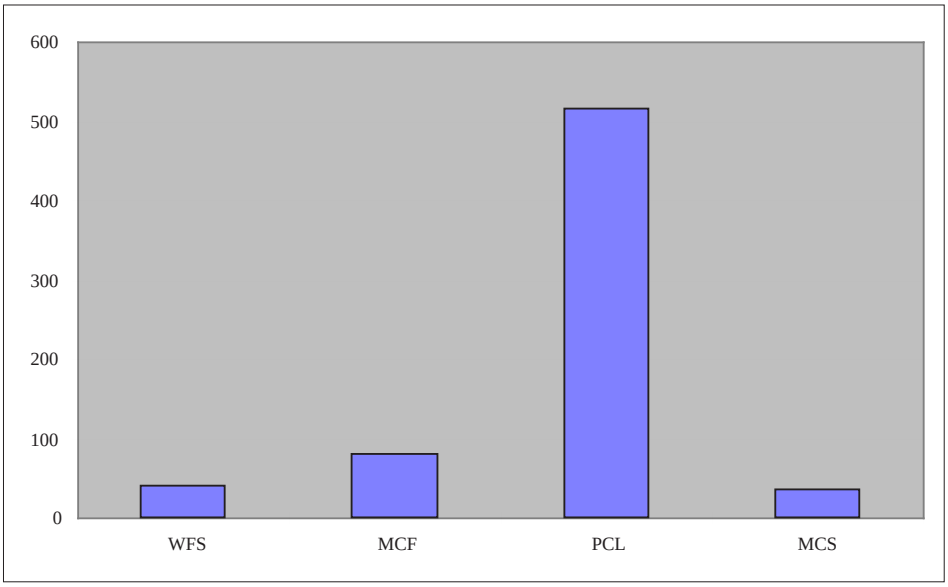


Abbildung 6.7: Vergleich des Speicherbedarfs von Objektsystemen.

6.4.2 Zweiter Vergleichstest

Um noch differenziertere Ergebnisse, vor allem beim Lesen und Schreiben von Slots zu erhalten, habe ich beim o. g. Test **ft4** und **ft4c** bei allen Systemen die Slots über Nachrichtenversenden gesetzt. Zusätzlich habe ich die Tests **ft3s** und **ft4s** durchgeführt, wobei statt Nachrichtenversenden das Konstrukt `slot-value` bzw. `get-slot` verwendet wurde. Verglichen wurden nur noch die Systeme PCL, MCF, MCS auf einem Apple Macintosh

[6] Aus dem gleichen Grund habe ich die Werte von **ft2** für das Balkendiagramm künstlich angenähert: bei MCF von ca. 79 Sekunden auf 30 Sekunden reduziert und den Wert bei MCS von ca. 0,17 auf 0,3 erhöht. Die Werte in der Tabelle B.1 sind natürlich unverändert.

II unter Allegro CommonLisp Version 1.2.1 sowie die Systeme MCF, MCS und Flavors auf einer SUN Workstation unter Lucid CommonLisp.

Die Abbildungen 6.8 und 6.9 zeigen die Meßergebnisse auf einem Apple Macintosh II in Allegro CommonLisp für die Objektsysteme PCL, MCF und MCS. Die genauen Zahlen befinden sich im Anhang B.2.

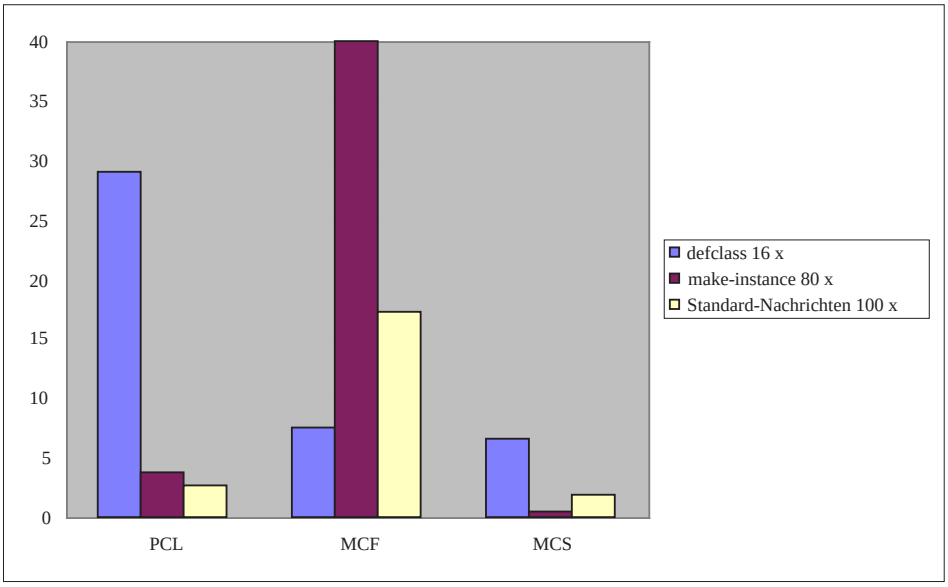


Abbildung 6.8: 2. Vergleich der Ausführungszeiten auf Apple Macintosh II in Allegro Common Lisp 1.2.1 in Sekunden, Teil 1.

Abbildung 6.10 und Abbildung 6.11 zeigen den entsprechenden Vergleich der Objektsysteme Flavors, MCF und MCS auf einer SUN Workstation in Lucid CommonLisp. Die genauen Zahlen befinden sich im Anhang B.3.

Interessant ist hier zu beobachten, daß die Meßergebnisse auf verschiedenen Rechnern in verschiedenen Lisp-Systemen teilweise stark abweichen. Insbesondere fällt auf, daß die Definition von Klassen in MCS 0.5 auf einer SUN Workstation unter Lucid CommonLisp dreimal länger dauert als auf einem Apple Macintosh II unter Allegro CommonLisp. Dies läßt sich einfach dadurch erklären, daß die Funktion `compile` in beiden Lisp-Systemen ein sehr unterschiedliches Verhalten aufweist. Auch wenn dieser Ausreißer unkritisch ist, weil Klassen in der Regel zur Ladezeit und nicht zur Ausführungszeit eines Programmms definiert werden, wurde er in der nächsten Version von MCS vermieden. Viel entscheidender ist die Effizienz bei der Erzeugung terminaler Instanzen, was sich insbesondere bei der Anwendung BABYLON noch zeigen wird.

Zusammenfassend läßt sich sagen, daß die getroffenen Design- und Implementierungsentscheidungen durch diese Meßergebnisse voll bestätigt wurden. MCS 0.5 zeigt signifikante Effizienzverbesserungen gegenüber vergleichbaren Objektsystemen und ist teilweise besser als kommerzielle Objektsysteme.

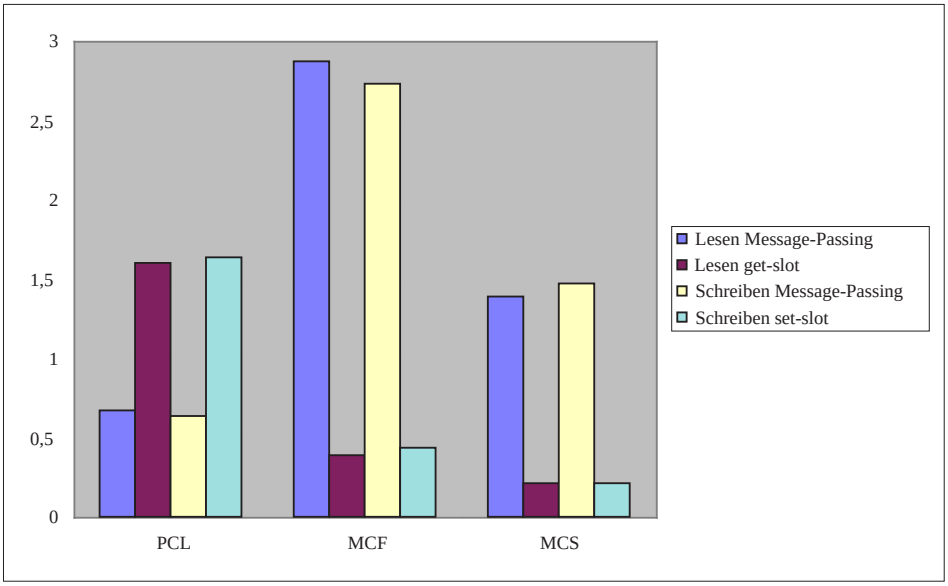


Abbildung 6.9: 2. Vergleich der Ausführungszeiten auf Apple Macintosh II in Allegro Common Lisp 1.2.1 in Sekunden, Teil 2.

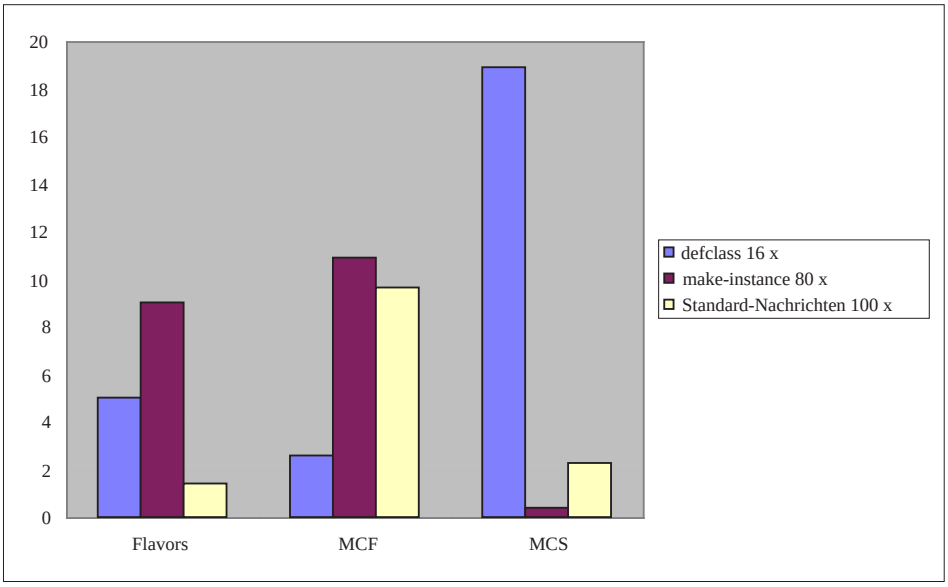


Abbildung 6.10: Vergleich der Ausführungszeiten auf SUN Workstation in Lucid Common Lisp in Sekunden, Teil I.

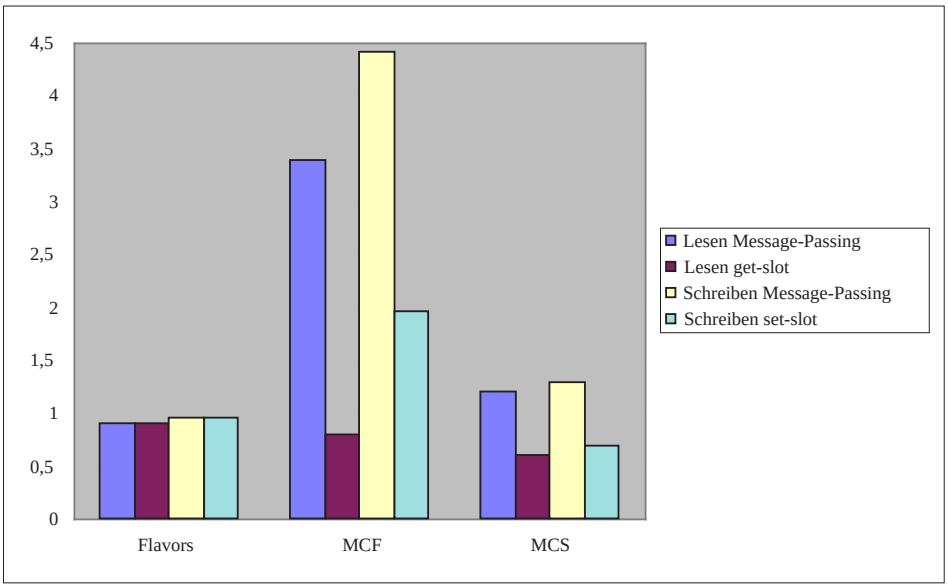


Abbildung 6.11: Vergleich der Ausführungszeiten auf SUN Workstation in Lucid Common Lisp in Sekunden, Teil II.

Natürlich weiß man aus Erfahrung, daß Benchmarks ihre Tücken haben, und deshalb hinterfragt werden müssen. Insbesondere läßt sich aus Einzelergebnissen noch kein gesichertes Verhalten eines Gesamtsystems ableiten. Schlimmer noch: die Benutzung komplexer Softwaresysteme ist grundsätzlich subjektiv. Ebenso subjektiv ist dementsprechend die Einschätzung ihrer Performanz. Meine Konsequenz aus diesem Dilemma ist die Evaluierung des Objektsystems an Hand einer realen Anwendung. Diese wird im nächsten Abschnitt beschrieben.

6.5 Reale komplexe Anwendung: BABYLON 2.1

Mit BABYLON wurde MCS 0.5 unter realen Bedingungen der Entwicklung und der Nutzung eines komplexen wissensbasierten Softwaresystems getestet. In Kapitel 2.1.1 habe ich dieses Anwendungsszenario ausführlich diskutiert. Einige Eckwerte seien hier in Erinnerung gerufen. Das gesamte System ohne die Objekte einer Wissensbasis umfaßt:

- 149 Klassendefinitionen,
- 596 Operationen (unterschiedliche Methodennamen) und
- 899 Methodendefinitionen.

Es ist zwar klar, daß die Komplexität von BABYLON weniger im Umfang seines Quellcodes zu suchen ist, als vielmehr in der kognitiven Komplexität bei der Integration unterschiedlicher Wissensrepräsentationsformalismen. Aus Sicht des Objektsystems sind die genannten Zahlen aber durchaus interessant, weil sie auch Schlüsse auf andere Anwendungen erlauben. Nicht nur die Effizienz, sondern auch die Robustheit eines Objektsystems kann dabei unter Beweis gestellt werden.

Zunächst gehe ich im nächsten Unterabschnitt auf die Implementierung von BABYLON-Flavors mit MCS 0.5 ein. Anschließend werden die Performanzmessungen für BABYLON unter den Objektsystemen MCF und MCS vorgestellt und diskutiert.

6.5.1 Implementierung von BABYLON-Flavors

Die Sprachkonstrukte von BABYLON-Flavors wurden im Abschnitt 6.1.1 bereits vorgestellt. Sie weichen geringfügig von den MCS-Konstrukten ab. Im wesentlichen wird nur ein Teil der MCS-Funktionalität benötigt. An einigen Stellen, muß die Funktionalität aber auch erweitert werden.

Die meisten Unterschiede in den Konstrukten von BABYLON-Flavors und MCS sind syntaktischer Natur. Die Implementierungsaufgabe besteht also im wesentlichen in der syntaktischen Abbildung der Konstrukte von BABYLON-Flavors auf MCS-Konstrukte. Dies kann durch entsprechende Makrodefinitionen umgesetzt werden. Die wichtigsten Konstrukte werden hier vorgestellt.

Klassen aus BABYLON-Flavors werden als spezielle MCS-Klassen realisiert. Es wird also eine neue Metaklasse mit dem Namen `flavor-class` definiert:

```
(defmetaclass flavor-class (req-inst-vars) ())
```

Im Unterschied zu Standardklassen in MCS benötigt man hier einen weiteren Slot `req-inst-vars` für die Klassen-Option `:required-instance-variables`. Diese Option ist erforderlich, wenn man auf Instanzvariablen in Methoden wie auf Lisp-Variablen zugreifen möchte, also über den Bezeichner. Entsprechende Vorkommen freier Variblen in Methodenrümpfen müssen vom Objektsystem dann ersetzt werden, wenn es sich um Instanzvariablen handelt. Zur Methodendefinitionszeit müssen also alle Instanzvariablen bekannt sein. Da man schon in generellen Methoden auf Instanzvariablen der Subklassen zugreifen will, muß man diese in der Klassenoption `required-instance-variables` deklarieren. Meiner Meinung nach ist dies ein zweifelhaftes Konzept. Es wird daher von MCS nicht direkt unterstützt. Da es in BABYLON aber verwendet wurde, muß es an dieser Stelle als Erweiterung bereitgestellt werden.

Anschließen kann das Konstrukt `def$flavor` realisiert werden. Das entsprechende Makro muß dafür sorgen, daß die neue Klassenoption `required-instance-variables` behandelt wird und daß ein Klassenobjekt als Instanz der Klasse `flavor-class` erzeugt und initialisiert wird.

An dieser Stelle wird demonstriert, wie eine einfach erscheinende Erweiterung die Implementierung deutlich erschwert. Die Ersetzung freier Variablen durch Slotreferenzen in Methodenrümpfen muß zur Übersetzungszeit erfolgen. Dazu braucht man alle Slots einer Klasse, einschließlich der ererbten. Daraus ergibt sich zwingend die Notwendigkeit, Klassen auch zur Übersetzungszeit zu erzeugen und zumindest teilweise zu initialisieren. Um überflüssige Berechnungen zur Übersetzungszeit zu vermeiden, wird die Initialisierung von Klassen in zwei Phasen aufgeteilt: `:basic-init` und `:init`. Die Methode `:basic-init` berechnet nur die Klassenpräzedenzliste und führt die Vererbung der Slots durch, während `:init` die restlichen Aktionen aus der Methode `:init` von `standard-class` ausführt:

```
(defmethod (flavor-class :basic-init) ()
  (declare (optimize (speed 3) (safety 0)))
   (send-self :compute-cplist)
   (send-self :inherit-slots-with-defaults)
   self)

(defmethod (flavor-class :init) (&rest inits)
  (declare (optimize (speed 3) (safety 0)) (ignore inits))
   (send-self :compute-slot-accessor-fn)
   (send-self :extend-subclasses-of-supers)
   (send-self :compute-slot-access-methods)
   (send-self :compute-basicnew-fn)
   self)
```

Die Konstruktorfunktion von `flavor-class` führt nur die Methode `:basic-init` aus, so daß `:init` außerhalb der Konstruktorfunktion aufgerufen wird.

Um zwischen dem Übersetzungs-, Lade- und Interpretationskontext unterscheiden zu können, gibt es in CommonLisp das Konstrukt `eval-when` mit den Optionen `:compile-top-level`, `:load-top-level` und `:execute` bzw. `compile`, `load` und `eval` in CLtL1. Seine Verwendung verhindert jedoch im allgemeinen die Komplettkompilation. In späteren Versionen von MCS wird `eval-when` daher nicht mehr benutzt. Hier sorgt es dafür, daß

Klassen auch zur Übersetzungszeit erzeugt und teilweise initialisiert werden. Zur Ladezeit wird darüberhinaus die vollständige Initialisierung angestoßen. Und im Interpretationsmodus wird zusätzlich der Programmentwicklungsaspekt berücksichtigt, um das Redefinieren von Klassen zur Entwicklungszeit zu unterstützen.

```
(defmacro def$flavor (a_class a_list-of-instance-variables
                              a_list-of-superclasses &rest options)
  `(progn
     (eval-when (:compile-toplevel)
       (defvar ,a_class)    ; to avoid compiler warnings
       (setq ,a_class
             (funcall (mcs-slot-value flavor-class (index-of-basicnew-fn))
                      flavor-class
                      :name ',a_class
                      ...
                      :req-inst-vars
                      ',(required-instance-variables options))))
     (eval-when (:load-toplevel)
       (setq ,a_class
             (send-fast (funcall (mcs-slot-value flavor-class
                                                 (index-of-basicnew-fn))
                                 flavor-class
                                 :name ',a_class
                                 ...
                                 :req-inst-vars
                                 ',(required-instance-variables options))
                        :init)))
     (eval-when (:execute)
       (let ((new-class
              (funcall (mcs-slot-value flavor-class
                                       (index-of-basicnew-fn))
                       flavor-class
                       :name ',a_class
                       ...
                       :req-inst-vars
                       ',(required-instance-variables options))))
         (if (flavorp ',a_class)
           (redefine-class ,a_class new-class)
           (setq ,a_class (send-fast new-class :init)))))))
```

Wird eine Flavor-Klasse redefiniert, so werden entsprechende Teile der Vererbungshierarchie dem neuen Stand angepaßt. Das heißt:

- Die redefinierte Flavor-Klasse muß aus den Subklassenlisten ihrer ehemaligen Superklassen entfernt werden.
- Die Subklassen der redefinierten Flavor-Klasse müssen ebenso redefiniert werden.

- Die Instanzen von geänderten Flavor-Klassen bleiben unverändert, müssen also vom Programmierer selbst angepaßt werden. Programmteile, die Instanzen erzeugen bzw. verwenden müssen erneut ausgewertet werden. Das Objektsystem weist durch eine entsprechende Warnung darauf hin.

Das Konstrukt `def$method` wird auf das MCS-Konstrukt `defmethod` abgebildet. Die Zusatzaufgabe besteht darin, die Slotreferenzen und `($send self ...)` Ausdrücke in Methodenrümpfen zu ersetzen. Wichtig ist, daß diese Quellcode-Analyse zur Makroexpansionszeit und somit zur Übersetzungszeit stattfindet. Die Voraussetzung dafür ist die Kenntnis aller Slots der betroffenen Klassen.

```
(defmacro def$method ((class-name . type&selector) varlist . body)
  (let ((new-body  ; wird zur Übersetzungszeit berechnet
         (compile-slot-references
           ;; die Klasse muß zur Übersetzungszeit existieren
           (get-all-required-slot-names  (symbol-value class-name))
           (subst-$send-self body))))
   `(defmethod (,class-name ,@type&selector) ,varlist ,@new-body)))
```

Hier wird allerdings nur eine einfache textuelle Analyse und Ersetzung durchgeführt, ohne angewandte Vorkommen von Makros zu expandieren etc. Sonst müßten unzählige Feinheiten der komplizierten Semantik von CommonLisp berücksichtigt werden. Diese Einschränkung stellte für die Anwendung BABYLON aber kein Problem dar.

Das Konstrukt `compile-$flavor-$methods` muß in MCS nicht unbedingt implementiert werden, weil Methoden ohnehin kompiliert und benötigte Kombinationen im Cache eingetragen werden. Auf der anderen Seite kostet das dynamische Caching wertvolle Laufzeit, während es zur Ladezeit weniger kritisch ist. `compile-$flavor-$methods` wurde daher auch bereitgestellt.

Die Konstrukte des Nachrichtenversendens können einfach auf das MCS-Konstrukt `send-message` abgebildet werden:

```
(defmacro $send (object message &rest args)
  `(send-message ,object ,message ,@args))

(defmacro lexpr-$send (object message &rest args)
  `(apply #'send-message ,object ,message ,@args))
```

Bei der obigen Ersetzung von `$send self` Ausdrücken in Methodenrümpfen wird aber auf das effizientere MCS-Konstrukt `send-self` abgebildet.

Auch die Implementierung von `make-$instance` kann durch eine einfache Makrodefinition erledigt werden:

```
(defmacro make-$instance (flavor &rest initializations)
  `(make-instance ,flavor ,@initializations))
```

Weitere Details der Realisierung von BABYLON-Flavors kann man im Quellcode unter (URL: xxx) finden. Insgesamt wurden die Implementierungs-Richtlinien aus Kapitel 5 schon weitgehend beachtet. Wenige Ausnahmen stellen die verwendeten Konstrukte aus CommonLisp, wie `eval-when` und `symbol-value` dar.

6.5.2 Performanzmessungen

Für die Performanzmessungen des BABYLON-Gesamtsystems wurden nur noch die Objektsysteme MCF und MCS in Betracht gezogen, weil die Einzeltests schon genügend aussagekräftig waren, um eine Vorauswahl treffen zu können. Hier wurde ein Vergleich der verschiedenen BABYLON-Prozessoren für konkrete Wissensbasen vorgenommen:

- `frames1` Für den Frame-Prozessor wurden die Zugriffe auf alle Slots aller Instanzen der Wissensbasis Pilze [Müller, 1986] gemessen.
- `frames1c` Test von `frames1` nach `compile-$flavor-methods`.
- `frames2` Es wurden 100 Zugriffe auf einen Slot einer Instanz der Wissensbasis Pilze gemessen.
- `frames2c` Test von `frames2` nach `compile-$flavor-methods`.
- `frames3` Es wurden 100 Zugriffe auf einen Slot einer Instanz der Wissensbasis Pilze gemessen.
- `frames3c` Test von `frames2` nach `compile-$flavor-methods`.
- `rules1` Für den Regel-Prozessor wurden 50 Regeln einer Wissensbasis gefeuert.
- `rules1a` Test von `rules1` mit anderem Algorithmus im Regel-Prozessor.
- `rules2` Für den Regel-Prozessor wurden 200 Regeln einer Wissensbasis gefeuert.
- `rules2a` Test von `rules2` mit anderem Algorithmus im Regel-Prozessor.
- `prolog1` Für den Prolog-Prozessor wurde die naive Umkehrung einer 30-elementigen Liste einer Wissensbasis berechnet.
- `prolog2` Ergebnis von `prolog1` in *lips*.

Die vollständigen Ergebnisse obiger Tests findet man im Anhang B.4. Abbildung 6.12 zeigt die Ergebnisse der Tests `frames3` und `frames3c`. Abbildung 6.13 zeigt die Ergebnisse der Tests `rules2` und `rules2a`. Ebenso wurde der Hauptspeicherbedarf von BABYLON unter den beiden Objektsystemen ermittelt. Mit MCS benötigte das BABYLON-Gesamtsystem 30 % weniger Hauptspeicher als mit MCF.

Betrachtet man die Vergleichswerte nach der Kompilation (`compile-$flavor-methods`), so ist der Frame-Prozessor durch MCS bis zum Faktor 2 schneller geworden. Der Regel-Prozessor ist 9 bis 12 mal schneller und der Prolog-Prozessor über 100 mal schneller geworden.

Um die Performanz des Prolog-Prozessors besser einschätzen zu können, wurde zusätzlich ein Vergleich mit einer in Lisp implementierten WAM (Warren Abstract Machine) [Warren, 1983], [Aït-Kaci, 1991] durchgeführt. Wie Abbildung 6.14 und Abbildung 6.15 zeigen, hat MCS für den Prolog-Prozessor so einen Effizienzsprung bewirkt, daß seine geplante

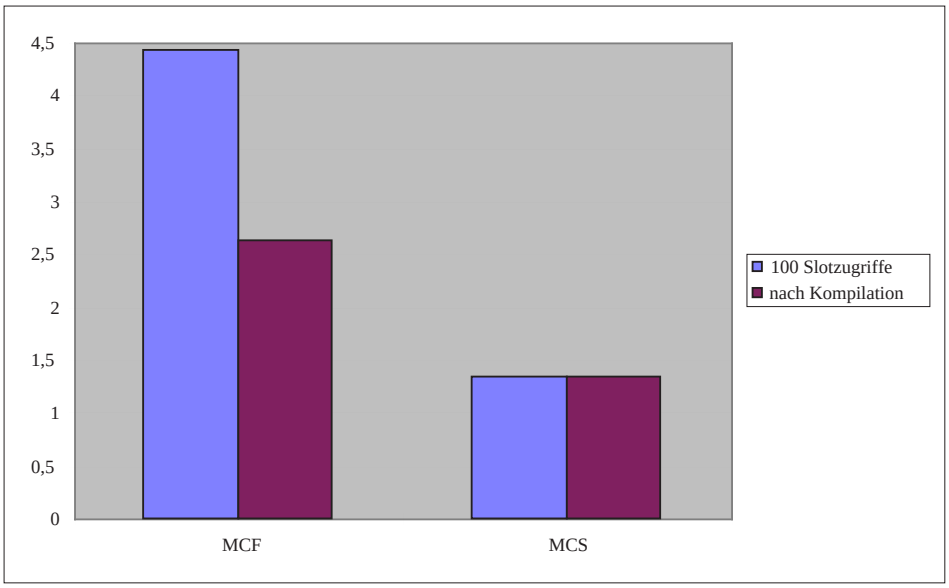


Abbildung 6.12: Vergleich der Ausführungszeiten des BABYLON-Frame-Prozessors in Sekunden.

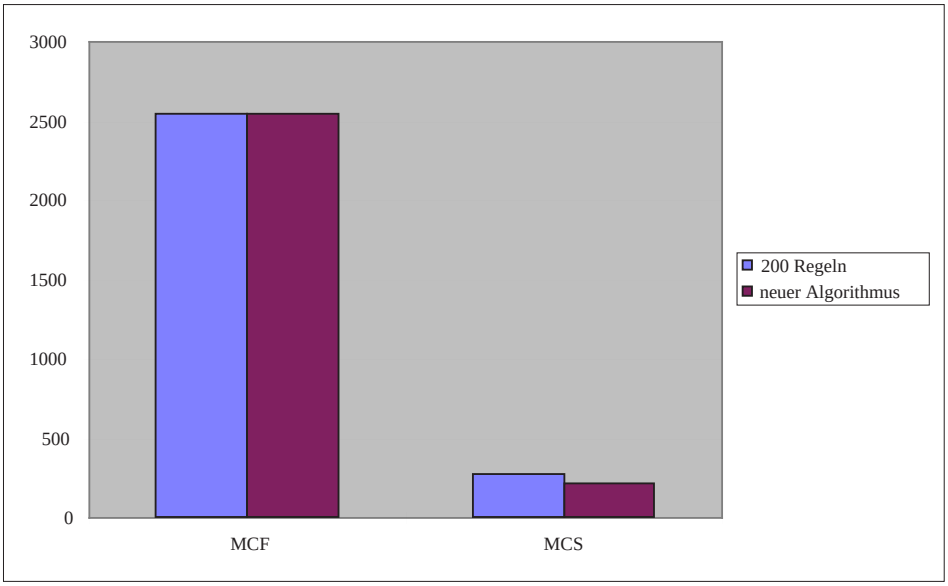


Abbildung 6.13: Vergleich der Ausführungszeiten des BABYLON-Regel-Prozessors in Sekunden.

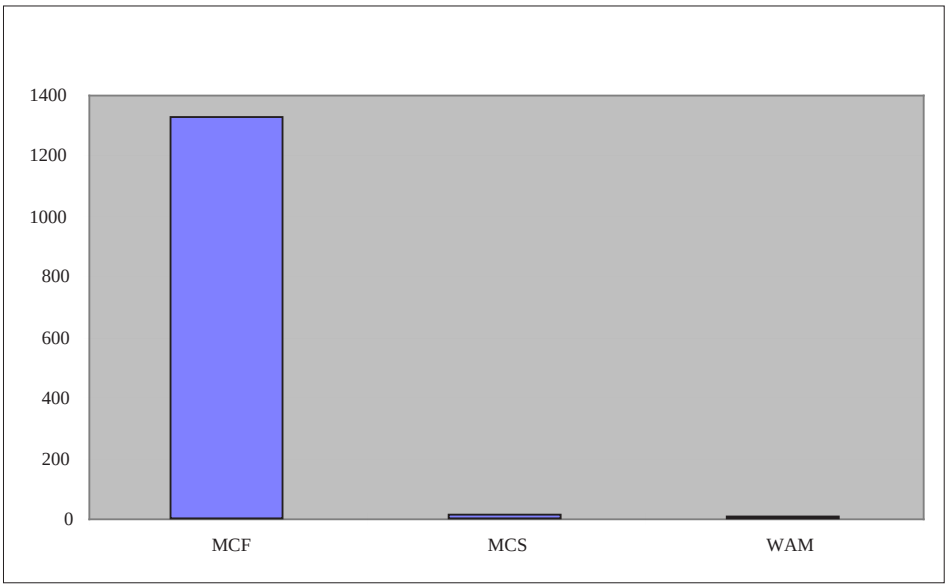


Abbildung 6.14: BABYLON-Prolog-Prozessor im Vergleich zur WAM in Sekunden.

Reimplementierung in einer WAM überflüssig wurde. Der Abstand zur WAM liegt mit MCS unter einem Faktor von 2. Die genauen Werte findet man im Anhang B in den Tabellen B.5 und B.6.

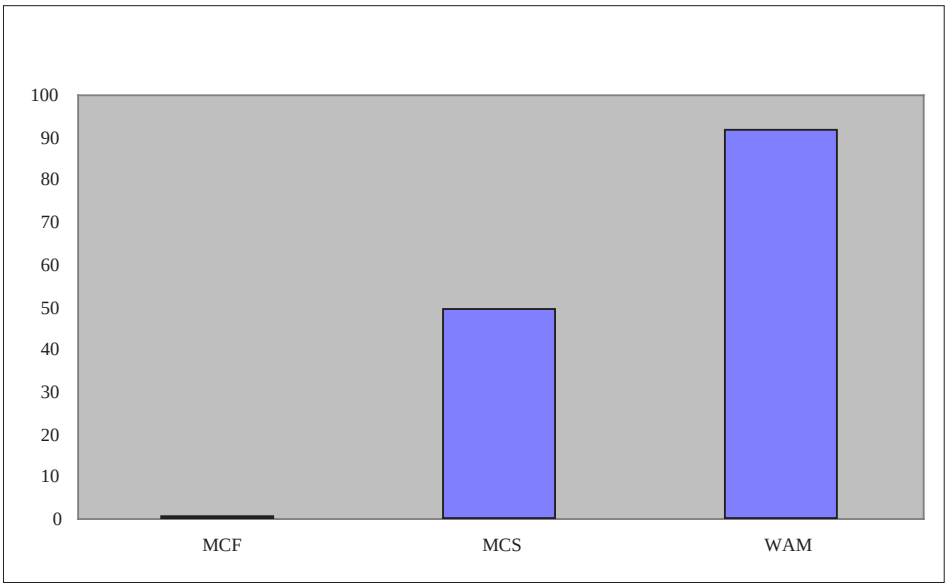


Abbildung 6.15: BABYLON-Prolog-Prozessor im Vergleich zur WAM in lips.

Die Relevanz der erzielten Effizienzresultate wird auch durch eine Umfrage bei den BABYLON-Anwendern untermauert [Christaller *et al.*, 1989, S. 448]. Diese Umfrage bezog sich auf die Versionen 0.0, 1.0, 1.1 und 2.0 von BABYLON auf verschiedenen Rechnerplattformen ohne MCS 0.5. Auch wenn die Laufzeit insgesamt als akzeptabel bewertet wurde, bemängelten alle Anwender die Ineffizienz des Prolog-Prozessors. Mit der Umstellung auf MCS 0.5 in der Version 2.1 und der einhergehenden Beschleunigung um den Faktor über hundert wurde auch hierfür eine überzeugende Lösung gefunden.

6.6 Bewertung von MCS 0.5

In diesem Abschnitt wird MCS in seiner ersten Realisierung zum einen im Hinblick auf die speziell hierfür gesetzten Kriterien und zum anderen im Hinblick auf die generelle Sicht objektorientierter Sprachkonzspte aus Kapitel 4 bewertet. Zunächst gehe ich auf Aspekte des Sprachdesigns, dann auf Aspekte der Implementierung ein. Zum Schluß wird das Ergebnis an den Anforderungen einer komplexen Anwendung gemessen.

6.6.1 Sprachdesign

Ausgangspunkt des Sprachdesign war das Objektsystems Flavors bzw. BABYLON-Flavors. Letzteres stellte im wesentlichen die Anforderungen an die Funktionalität von MCS. Die reflektive Systemarchitektur folgt weitgehend dem Konzept von ObjVLisp.

Entsprechend den in Abschnitt 6.2.1 formulierten Zielen erfüllen die Sprachkonstrukte von MCS 0.5 weitgehend die Kriterien Minimalität, Reflektivität, Erweiterbarkeit, Uniformität, Orthogonalität, Effizienz und Portabilität.

So konnte die Schnittstelle BABYLON-FLAVORS einfach als Erweiterung des MCS-Kerns realisiert werden. Allerdings wurde dabei auch deutlich, daß das bereitgestellte Metaobjektprotokoll einige wichtige Aspekte des Objektsystems nicht berücksichtigt. Nur Klassen sind als Metaobjekte spezifiziert, nicht aber Feldannotationen, generische Operationen bzw. Methoden. Die Spezialisierung des Feldzugriffsverhaltens sowie der Speicherrepräsentation von Objekten wird nicht unterstützt. Will man an diesen Stellen Spezialisieren, muß man auf die Implementierungsebene von MCS absteigen. Das Metaobjektprotokoll ist an einigen Stellen zu grobkörnig. Es fehlt z. B. die Methode, um nicht initialisierte Objekte zu erzeugen.

MCS 0.5 unterstützt nur allgemeine multiple Vererbung nach der Strategie *depth-first-left-to-right-up-to-joins*. Einfache Vererbung und Mixin-Vererbung können zwar als Erweiterung von MCS realisiert werden. Diese Vorgehensweise erscheint aber nicht natürlich, weil man in der Erweiterung eine Einschränkung der Funktionalität vornimmt. Ein weiterer Nachteil ist, daß das kompliziertere Konzept zum Regelfall bestimmt wurde.

Die genannten Defizite von MCS 0.5 folgen aber nicht aus der gewählten Systemarchitektur, sondern liegen in der Natur eines ersten Entwurfs begründet. Sie zeigen auf, welche weiteren Anforderungen in der nächsten Version von MCS aufgegriffen und erfüllt werden müssen.

Die gewählte Systemarchitektur hat wesentlich dazu beigetragen, eine effiziente Implementierung zu ermöglichen und somit die Hauptdefizite bisheriger Objektsysteme aus Abschnitt 6.1.2 zu überwinden.

6.6.2 Implementierung

Insgesamt haben sich die getroffenen Implementierungsentscheidungen bewährt. Die gewählte Objektrepräsentation ist mit Abstand die effizienteste bzgl. Speicherbedarf, Erzeugungs- und Zugriffszeit. Auch das Prinzip, kostenträchtige Berechnungen in die Klasseninitialisierungsphase zu verlagern (`:compute-slot-accessor-fn` und `:compute-basicnew-fn`) hat signifikante Effizienzverbesserungen bewirkt. Insbesondere die Objekterzeugung wurde um Größenordnungen schneller als bei vergleichbaren Objektsystemen. Nachteilig hat sich in diesem Zusammenhang die Verwendung des inkrementellen Compilers von COMMONLISP durch Aufruf der Funktion `compile` erwiesen. In einigen Lisp-Systemen ist der inkrementelle Compiler zu langsam für solche Zwecke, obwohl das berechnete Ergebnis dann die maximale Effizienz bietet. So ist die relativ langsame Klassenerzeugung auf SUN Workstations unter Lucid Common Lisp zu erklären. Ein weiterer Nachteil der Verwendung von `compile` ist der Umstand, daß MCS 0.5 so nicht komplettkompilierbar ist.

Eine Schwäche hat sich auch beim Nachrichtenversenden gezeigt. Unter den gesetzten Randbedingungen, z. B. daß `send-message` eine Funktion sein mußte und kein Makro, bleiben Methodenaufrufe beim Message-Passing langsamer als Aufrufe generischer Funktionen wie in PCL. Daraus ergibt sich die Konsequenz, in späteren Versionen von MCS auf das Konzept generischer Funktionen zu setzen.

6.6.3 Zusammenfassung

Aus Sicht von BABYLON als einer komplexen Anwendung hat MCS alle Anforderungen erfüllt. Die erzielten Effizienzsteigerungen für BABYLON haben die Erwartungen bei weitem übertroffen. Im nachhinein, fragt man sich, warum die Vorgängersysteme so ineffizient waren. Die prinzipiellen Gründe liegen in der Quellcode-Interpretation zur Ausführungszeit, deren Notwendigkeit mit MCS widerlegt wird.

Die Objektebene von MCS 0.5 hat sich weitgehend an vorausgegangenen Objektsystemen orientiert. Die reflektive Systemarchitektur unterstützte eine effiziente Implementierung des Objektsystems selbst. Das einfache Metaobjektprotokoll war angemessen, um die Schnittstelle BABYLON-Flavors als Spezialisierung von MCS schnell zu realisieren. Die Performanz von MCS hat den Durchbruch für BABYLON als einer komplexen Anwendung auf Rechnern mit geringerer Leistungsfähigkeit als auf den Lisp-Maschinen ermöglicht. Die noch vorhandenen Defizite werden in den nächsten Kapiteln mit MCS 1.0 und schließlich mit Τελος schrittweise beseitigt.

Kapitel 7

Die Weiterentwicklung: MCS als kompatible Verbesserung von CLOS

Bach walked 60 miles to copy Buxtehude's music ...

Frederick P. Brooks, Jr. 1993. Keynote address: Language Design as Design, The Second History of Programming Languages Conference (HOPL-II), [Brooks, 1996, S. 10].

Ausgehend von den Ergebnissen und Erfahrungen mit MCS 0.5 wird in diesem Kapitel ein zweites Objektsystem vorgestellt. Statt des Nachrichtenversendens wird hier das Konzept der generischen Funktionen mit mehrfachem Methoden-Dispatch realisiert. Der metareflektive Kern wird erweitert. Nun werden auch Feldannotationen sowie generische Funktionen und Methoden zum Gegenstand der Reflektion, d. h. zu Objekten erster Ordnung. Das Konzept der Mixin-Vererbung wird bereits im Kern bereitgestellt und zur Realisierung von MCS 1.0 selbst verwendet.

Funktional orientiert sich MCS 1.0 an CLOS auf der Objektebene sowie an CLOS MOP auf der Metaobjektebene, siehe Kapitel 3.4.3. Es werden jedoch konzeptionell und implementierungstechnisch begründete Vereinfachungen vorgenommen. Dadurch wird sowohl der objektorientierte Programmierstil als auch die Effizienz komplexer Anwendungen verbessert. Insbesondere trägt das Konzept der Mixin-Vererbung hierzu bei. Ebenso positiv wirkt sich aus, daß Aspekte der Programmentwicklung identifiziert und als objektorientierte Erweiterungen des Kerns bereitgestellt werden. Redefinieren von Klassen und automatische Anpassung aller abhängigen Objekte ist somit nicht mehr der Regelfall, sondern eine bewußte Ausnahme, jedenfalls zur Ausführungszeit. Ein globaler Systemschalter erlaubt, auf einfache Weise festzulegen, ob redefinierbare Klassen als Standard-Metaobjekte zum Tragen kommen, z. B. generell zur Entwicklungszeit, oder die einfacheren und effizienteren Metaobjekte, insbesondere zur Auslieferungzeit von Applikationen (engl. *release version*).

Um Wiederholungen zu vermeiden, konzentriere ich meine Darstellung von MCS 1.0 auf die Unterschiede zu CLOS bzw. zu MCS 0.5. Es werden daher auch nur die hinzukom-

menden bzw. abweichenden Design- und Implementierungsaspekte im Vergleich zu MCS 0.5 behandelt. Eine vollständige Beschreibung von MCS 1.0 findet man in [Bretthauer und Kopp, 1991]. Die erfolgreiche Portierung von MCS nach Scheme wird in [Lange, 1993] ausführlich behandelt.

7.1 Diskussion von CLOS

Eine ausführliche Darstellung von CLOS wurde bereits in Kapitel 3.4.3 gegeben. Dabei habe ich bereits auf die Vor- und Nachteile bestimmter Sprachkonzepte im Vergleich zu anderen Sprachen hingewiesen. Hier möchte ich die problematischen Sprachkonstrukte nochmal zusammenfassen und begründen, warum in MCS 1.0 entsprechende Vereinfachungen vorgenommen wurden.

Zu Beginn der Entwicklung von MCS 1.0 gab es keine vollständige Implementierung von CLOS. Es lag nur ein Entwurf der CLOS-Spezifikation vor, die in Portable Common Loops PCL bei Xerox PARC schrittweise implementiert wurde. Die wichtigsten Sprachkonstrukte von CLOS waren jedoch in PCL schon früh verfügbar. Sie reichten insbesondere aus, um den Performanzvergleich mit MCS 0.5 durchzuführen, siehe Kapitel 6.4. Wie wir gesehen haben, lagen die Performanzschwächen von PCL vor allem in seinem Speicherbedarf und der deutlich langsameren Objekterzeugung im Vergleich zu MCS 0.5. Die Gründe dafür liegen im wesentlichen im Sprachdesign von CLOS, das keine einfache und effiziente Implementierung des Systems zuläßt. Akzeptable Performanz kann nur durch aufwendige dynamische Optimierungen erreicht werden. Diese greifen oft auch nur im Standardfall, nicht aber falls MOP-Operationen tatsächlich spezialisiert werden, wofür sie ja eigentlich bereitgestellt werden.

Zwei Schwachstellen im Sprachdesign von CLOS werden im folgenden genauer diskutiert: die Erzeugung und Initialisierung von Objekten als Beispiel für die Objektebene und das Slotzugriffsprotokoll mit stärkerem Bezug zur Metaobjektebene. Probleme allgemeiner multipler Vererbung wurden schon in Kapitel 4 auf Seite 90 sowie in [Bretthauer *et al.*, 1989c] und [Kopp, 1996a, S. 25] ausführlich diskutiert. Auch die Problematik von *shared slots* wurde bereits in [Kopp, 1996a, S. 21ff und 41] behandelt. Daß die Entbindung von der Pflicht, generische Funktionen mit `defgeneric` definieren zu müssen, bevor man entsprechende Methoden mit `defmethod` definiert, eine zweifelhafte Benutzerunterstützung darstellt, wurde schon in [Bretthauer *et al.*, 1994, S. 149] festgestellt. Als Konsequenz können Applikationen, die `defgeneric` einsparen, nicht komplettkompiliert werden.

7.1.1 Erzeugung und Initialisierung von Objekten in CLOS

Die Erzeugung und Initialisierung von Objekten in CLOS wurde in Kapitel 3.4.3 auf Seite 60 bereits im Überblick beschrieben und soll hier weiter vertieft werden. Das Grundkonzept von Flavors und CLOS, Objekte mit einem generellen Konstruktor `make-instance` angewandt auf ein Klassenobjekt und optionale Schlüsselwort-Argumente zu erzeugen, unterstützt objektorientiertes Programmieren nach meiner Einschätzung wesentlich besser als z. B. das Konzept von Java. In Java muß für jede Konstellation der Initialisierungsparameter je ein Konstruktor in der Klasse selbst und ggf. in allen Superklassen definiert werden. Auf den Quellcode der Superklassen hat man aber möglicherweise keinen Zugriff,

was dann zu undurchsichtigen *work arounds* führen kann. Worin liegt nun das Problem der CLOS-Lösung? Und wie kann man sie verbessern?

An der Erzeugung und Initialisierung von Objekten sind in CLOS vier generische Funktionen mit folgender Aufrufstruktur beteiligt:

```
(make-instance 'C :s1 1 :s2 2 ...)
  (make-instance (find-class 'C) :s1 1 :s2 2 ...)
    (allocate-instance (find-class 'C) :s1 1 :s2 2 ...)
    (initialize-instance new-object :s1 1 :s2 2 ...)
      (shared-initialize new-object T :s1 1 :s2 2 ...)
```

Alle vier Operationen können vom Benutzer schon auf der Objektebene spezialisiert werden und besitzen `&key`- und `&rest`-Parameter. Allein der *Overhead* durch den vierfachen generischen Dispatch sowie die aufwendigere Aufrufverwaltung bei `&key`- und `&rest`-Parametern ist zu hoch, um dieses dynamische Protokoll tatsächlich vollständig auszuführen. Dieses Protokoll ist somit von vornherein ein Scheinprotokoll, weil keine effiziente CLOS-Implementierung sich danach verhält. Wie sie sich wirklich verhält, bleibt für den Benutzer verborgen. Die Grundidee von Spracherweiterungsprotokollen, ein transparentes Implementierungsmodell der Objektebene offenzulegen, kann hier nicht eingelöst werden. Optimierungen beruhen auf der Kenntnis aller beteiligten Methoden. Relativ einfach zu optimieren ist der Spezialfall, wenn keine benutzerdefinierten Methoden für die beteiligten generischen Funktionen vorliegen. Die meisten CLOS-Implementierungen beschränken sich darauf, nur diesen zu behandeln. In allen anderen Fällen bleibt die Objekterzeugung dann nicht optimiert.

Ein weiteres Problem bei der Objekterzeugung und Initialisierung stellen die Initialisierungsschlüsselwörter dar. Wie das Beispiel 3.4.3 zeigte, können für einen Slot mehrere Initialisierungsschlüsselwörter spezifiziert werden. Default-Werte können für Slots über die Slotoption `:initform` sowie für die Schlüsselwörter als Klassenoption `:default-initargs` und in der `&key`-Parameter-Option der beteiligten generischen Funktionen spezifiziert werden. Dies kann die Benutzung sehr kompliziert machen. Wird tatsächlich von allen Möglichkeiten Gebrauch gemacht, so kann kaum noch nachvollzogen werden, welcher Default-Wert zum tragen kommt. Natürlich ist auch die Implementierung entsprechend aufwendig. Zum einen muß `make-instance` die explizit angegebenen Initialisierungsargumente um die sogenannten *defaulted* Initialisierungsargumente, die nicht explizit vorkamen, ergänzen. Dies erfordert einen polynomialen Zeitaufwand (Anzahl der expliziten Argumente mal Anzahl aller Defaults einer Klasse). Mehrere Initialisierungsschlüsselwörter pro Slot bewirken eine kubische Zeitkomplexität der Slot-Initialisierung an Stelle einer quadratischen bei höchstens einem Schlüsselwort pro Slot.

7.1.2 Slotzugriffe in CLOS

In CLOS gibt es zwei Alternativen, auf Slots zuzugreifen. Der abstraktere Zugriff erfolgt über die automatisch generierten Lese- und Schreiboperationen entsprechend den Slotoptionen `:reader`, `:writer` und `:accessor` in einer Klassendefinition. Da dies generische Funktionen sind, kann der Benutzer weitere Methoden über `defmethod` hinzufügen, z. B. Before-, After- oder Around-Methoden, und auch die generierte Primär-Methode

ersetzen. Die andere Möglichkeit, auf Slots zuzugreifen, bietet das implementierungsnähere Konstrukt `slot-value`. Die generierten Primärmethoden verhalten sich so, als ob sie `slot-value` aufrufen würden. Sie müssen es aber nicht unbedingt tun, um Systemoptimierungen zu erlauben. Denn `slot-value` ist selbst keine primitive Operation, sondern sie ruft die generische Funktion `slot-value-using-class` auf, wie in Kapitel 3.4.3 auf Seite 66 bereits beschrieben. Es ist also folgende Aufrufstruktur spezifiziert:

```
(read-slot-x object)
  (slot-value object 'x)
     (slot-value-using-class (class-of object)
                             object
                             (find-slot (class-of object) 'x))
```

Möchte der Benutzer das Zugriffsverhalten spezialisieren, so stellt sich das konzeptionelle Problem, daß er nicht weiß, an welcher Stelle er dies tun soll: bei der generischen Zugriffsfunktion auf der Objektebene oder bei `slot-value-using-class` auf der Metaobjektebene? Das offensichtliche Effizienzproblem dieses Designs ist, daß man eine spezielle Operation auf eine allgemeine abbildet und unterwegs die spezielle Information zunächst verliert, sie später aber wieder finden muß.

Wie schon die Erzeugung und Initialisierung von Objekten stellt auch das Slotzugriffsprotokoll in CLOS kein adäquates Implementierungmodell dar. Es kann nicht auf diese Weise effizient implementiert werden. Optimierungen können diese langen Aufrufkette abkürzen, falls keine benutzerdefinierten Methoden für `slot-value-using-class` vorliegen. Wird nachträglich eine solche definiert oder anwendbar auf Grund einer Klassenredefinition, müssen evtl. durchgeführte Optimierungen wieder zurückgenommen werden. Dazu ist eine aufwendige Abhängigkeitsverwaltung nötig.

Die aufgezeigten Schwachstellen im Design von CLOS im Hinblick auf Benutzung und Implementierung werden nun in MCS 1.0 teilweise beseitigt. Abschließende Lösungen werden erst mit ΤΕΛΟΣ präsentiert, das mit CLOS nicht mehr kompatibel sein mußte, wie MCS 1.0.

7.2 Entwurf und Implementierung von MCS 1.0

Der Entwurf von MCS 1.0 gründet auf der Arbeitshypothese, daß die objektorientierten Konzepte von CLOS im Prinzip richtig und wegweisend sind. Hierzu zählt die weitgehende Integration objektorientierter Konzepte mit der funktionalen Basissprache Lisp. Die Einführung generischer Funktionen an Stelle des Nachrichtenversendens unterstützt diese Integration. Ebenso die syntaktische Trennung von Klassendefinitionen von Definitionen generischer Funktionen bzw. Methoden. Dadurch können Lisp-Basistypen harmonisch in objektorientierte Programme eingebunden werden, ohne sie spezialisieren zu müssen, nur weil man ihr Verhalten erweitern möchte. Generische Funktionen können über Lisp-Basisklassen diskriminieren, im Unterschied zu JAVA.

Beim Design von CLOS wurde jedoch zu wenig auf die Orthogonalität, die Robustheit und die effiziente Implementierbarkeit der bereitgestellten Sprachkonstrukte geachtet. Neben den Zielen und Designkriterien für MCS 0.5 aus Kapitel 6.2.1 auf Seite 154, soll hier eine

weitgehende Kompatibilität, zumindest der Objektebene, zu CLOS eingehalten werden. Der Lösungsansatz besteht daher im wesentlichen in der Vereinfachung von CLOS. Entwurf und Implementierung hängen hier so eng zusammen, daß ich beides zusammenfasse. Eine getrennte sequentielle Darstellung wäre nicht sinnvoll.

7.2.1 Vereinfachungen gegenüber CLOS

Die wichtigsten Vereinfachungen gegenüber CLOS betreffen folgende Sprachelemente:

1. Mixin-Vererbung ist der Regelfall; allgemeine multiple Vererbung wird als Erweiterung realisiert.
2. Klassen müssen *top-down* definiert werden, d. h. die Superklassen müssen textuell vor den Subklassen definiert werden.
3. Redefinieren von Klassen wird als Erweiterung durch spezielle Metaklassen unterstützt, nicht von Standardklassen.
4. Es gibt pro Slot höchstens ein Initialisierungsschlüsselwort. Die Klassenoption `:default-initargs` entfällt.
5. Es werden nur klassenspezifische Methoden unterstützt, keine `eql`-Methoden, die in CLOS instanzspezifisch sind.
6. Die Parameterlisten von generischen Funktionen und Methoden enthalten nur geforderte Parameter und optional einen `&rest`-Parameter, wie schon in MCS 0.5.
7. *Multiple Werte* werden als Ergebnis von generischen Funktionen bzw. Methoden nicht unterstützt.
8. Das Konstrukt `call-next-method` läßt keine Angabe expliziter Argumente zu. Die nächstallgemeinere Methode wird immer mit denselben Argumenten aufgerufen, wie die rufende Methode.

Wie schon für einige Sprachaspekte erwähnt, spricht nichts dagegen, flexiblere und ausdrucksstärkere Sprachmittel als Erweiterung des Objektsystemkerns anzubieten. Es muß jedoch vermieden werden, daß auch Anwendungen, die mit einfachen Sprachmitteln auskommen, an Performanz und Robustheit einbüßen.

7.2.2 Sprachkonstrukte der Objektebene

Die Sprachkonstrukte der Objektebene von MCS 1.0 können im wesentlichen als eine Teilsprache von CLOS gesehen werden. Um aber die Mixin-Vererbung explizit zu unterstützen, werden zusätzlich zu `defclass` zwei weitere Konstrukte eingeführt: `defabstract` und `defmixin`. Unter Verwendung der Klassenoption `:metaclass` können die neuen Konstrukte auf das allgemeinere Konstrukt `defclass` zurückgeführt werden. Will man aber die Objekt- und die Metaobjektebene strikt voneinander trennen, so sollte es auf der Objektebene keine expliziten Verweise auf Metaklassen geben. Dies erreicht man durch die eingeführten Konstrukte `defabstract` und `defmixin`. Ihre Verwendung soll durch folgendes Beispiel verdeutlicht werden.

```
;;; In module A or file A.lisp
;;; define an abstract class of all graphical objects

(defabstract graphical-Object () ())

;;; define some methods for graphical objects
...

;;; In module B or file B.lisp
;;; define a base class Point

(defclass Point (graphical-Object)
  ((x :initarg :x
      :initform 0
      :accessor point-x)
   (y :initarg :y
      :initform 0
      :accessor point-y)))

;;; define some methods for point objects
...

;;; In module C or file C.lisp
;;; define a protocol for the color aspect of objects

(defconstant default-color 'grey)  ; a protocol constant

;;; the protocol operations
(defgeneric color-of (obj))
(defgeneric change-color (obj color))

;;; define the default behaviour of uncolored objects
(defmethod color-of (obj) default-color)
(defmethod change-color (obj color)
  (warn "Can't change the color of an uncolored object."))

;;; define the color aspect of colored object

;;; define the structure of the color aspect as a mixin class
(defmixin colored ()
  ((color :initarg color
          :initform 'black)))

;;; define the behavior of colored objects
(defmethod color-of ((obj colored))
   (slot-value obj 'color))
```

```
(defmethod change-color ((obj colored) color)
  (setf (slot-value obj 'color) color))

;;; In module D or file D.lisp
;;; define a protocol for the moving aspect of objects

;;; the protocol operations
(defgeneric move-object (obj))

;;; define the default behaviour of unmoveable objects
(defmethod move-object (obj) default-color)
  (warn "Can't move an unmoveable object."))

;;; define the moving aspect of moveable object

;;; define the structure of the moving aspect as a mixin class
(defmixin moveable ()
  (...))

;;; define the behavior of moveable objects
(defmethod move-object ((obj moveable))
   ...)


;;; In module E or file E.lisp
;;; build an application using reusable components

;;; define required classes
(defclass colored-Point (colored Point) ())
(defclass moveable-colored-Point (moveable colored Point) ())

;;; create some instances
(setf p1 (make-instance 'Point 'x 25 'y 10))
(setf p2 (make-instance 'colored-Point 'x 25 'color 'red))
(setf p3 (make-instance 'moveable-colred-Point 'y 10 'color 'blue))

;;; execute some operations on instances
...
```

Im Prinzip kann dieses Beispiel auch in CLOS so ähnlich formuliert werden, indem man statt der Konstrukte `defabstract` und `defmixin` einfach `defclass` verwendet. Dann bleibt aber die Intention, wie die Klassen verwendet werden sollen, nicht spezifiziert. In der Implementierung können keine entsprechenden Optimierungen sicher vorgenommen werden. Sie können nur auf Verdacht probiert werden, müssen aber bei anderer Verwendung wieder zurückgenommen werden.

Das Beispiel zeigt, daß generische und wiederverwendbare Softwarekomponenten in CLOS deutlich besser zu implementieren sind als in anderen objektorientierten Sprachen, wie

JAVA beispielsweise. Dort kann kein Default-Verhalten anderweitig definierter Objekte bzgl. eines neuen Verhaltensaspekts spezifiziert werden. In JAVA können nur sogenannte statische Methoden für andere Objekte definiert werden, die aber nicht spezialisiert werden dürfen. Sie sind somit nicht objektorientiert.

Wie die Mixin-Vererbung für das Design und die Implementierung von MCS 1.0 selbst verwendet wird, beschreibe ich in Abschnitt 7.2.5. In den folgenden Abschnitten gehe ich exemplarisch auf zwei Vereinfachungen gegenüber CLOS ein.

7.2.3 Vereinfachung der Erzeugung und Initialisierung von Objekten

Die Vereinfachung im Vergleich zu CLOS ergibt sich aus der Anwendung des Orthogonalitätskriteriums auf das Sprachdesign. Danach sollte auf der gegebenen Granularitätsebene jeder Aspekt durch genau eine generische Funktion spezialisierbar sein. Hier handelt es sich um zwei Aspekte. Die Objektallokation ist eine Klassenoperation, die auf der Metaobjektebene spezialisierbar ist. Die Objektinitialisierung ist eine Objektoperation und kann auf der Objektebene spezialisiert werden. Da man in der Regel Objekte sofort nach ihrer Erzeugung initialisiert gibt es eine nicht-generische Funktion `make-instance`, die beide generische Funktionen `allocate-instance` und `initialize-instance` aufruft.

```
(make-instance C :s1 1 :s2 2 ...)
  (allocate-instance C '(:s1 1 :s2 2 ...))
  (initialize-instance new-object :s1 1 :s2 2 ...)
```

Geht man davon aus, daß `allocate-instance` und `initialize-instance` im Programmtext sehr selten direkt aufgerufen werden, könnten beide ohne Rest-Parameter definiert werden. Wegen der geforderten Kompatibilität zu CLOS, vor allem auf der Objektebene, wurde für `initialize-instance` der Rest-Parameter beibehalten. Auf die generische Funktion `shared-initialize` kann ganz verzichtet werden, weil das Redefinieren von Klassen sowie `change-class` im Standardfall nicht unterstützt wird.

Damit wurde der organisatorische *Overhead* im Vergleich zu CLOS mindestens halbiert. Die algorithmische Vereinfachung bei der Initialisierung ergibt sich aus dem Verzicht auf die Klassenoption `:default-initargs` sowie aus der Beschränkung auf höchstens ein Initialisierungsschlüsselwort pro Slot.

Selbst wenn für eine CLOS-Klasse keine `:default-initargs` spezifiziert wurden, bleibt der *Overhead* bestehen, dieses zur Ausführungszeit festzustellen. Im allgemeinen kostet diese Feststelung einen generischen Funktionsaufruf und einen Identitätsvergleich. Sind `:default-initargs` spezifiziert worden, so muß `make-instance` zunächst eine neue Liste aus den expliziten Argumenten und der vorberechneten Liste aller *Default-Initargs* erzeugen. Selbst wenn Listenerzeugung in Lisp optimiert ist, werden dadurch viele temporäre Listen erzeugt und somit die automatische Speicherbereinigung belastet. Diese Probleme werden in MCS 1.0 von vornherein vermieden.

Die Möglichkeit mehrerer Initialisierungsschlüsselwörter pro Slot zwingt in CLOS, die Initialisierung der Slots in zwei Phasen durchzuführen. Zuerst müssen die Initialisierungsargumente abgearbeitet werden. Dabei muß die Zulässigkeit der Initialisierungsschlüsselwörter geprüft und die entsprechenden Slots so gesetzt werden, daß die

Schlüsselwörter von links nach rechts zum tragen kommen, falls mehrere Initialisierungsargumente denselben Slot setzen würden. Die schnellste Lösung dafür ist, die Initialisierungsargumente von rechts nach links abzuarbeiten, auch wenn dadurch ein und derselbe Slot evtl. mehrmals gesetzt werden muß. Das am weitesten links stehende Argument wird somit Vorrang haben. Man erhält einen aufwändigen Algorithmus mit dreifach geschachtelten Schleifen:

```
   ;; first phase
11 (doplist ((supplied-key supplied-value) initargs)
     ;; check if legal supplied-key
12   (... (member supplied-key legal-keywords) ...)
     ;; maybe initialize a slot
13   (dolist (slot slots)
14     (dolist (slot-key slot-keywords)
15       (if (eq supplied-key slot-key)
            ;; set slot according to supplied value
16          ...)
17       ...) ...) ...)
```

Die Anzahl der Iterationsschritte ergibt sich aus der Multiplikation der Anzahl der expliziten Initialisierungsargumente mit der Anzahl der Slots und der Anzahl der Schlüsselwörter pro Slot.

In einer zweiten Phase müssen alle Slots durchlaufen werden. Dabei muß für jeden Slot geprüft werden, ob er schon initialisiert wurde. Falls nicht, wird er gemäß der Slotoption `:initform` initialisiert.

```
   ;; second phase
21 (dolist (slot slots)
22   (if (slot-unbound-p slot ...)
       ;; set slot according to :initform
23     ...))
```

In MCS 1.0 können die Schritte 13 bis 17 aus der ersten Phase zum Teil ganz eingesprart werden und zum Teil (Schritt 16) während der zweiten Phase erledigt werden:

```
   ;; first phase
11 (doplist ((supplied-key supplied-value) initargs)
     ;; check if legal supplied-key
12   (... (member supplied-key legal-keywords) ...))
   ;; second phase
21 (dolist (slot slots)
     ;; look for a supplied initarg value
22a  (if (getf (slot-keyword slot) initargs ...)  ; just one keyword
       ;; set slot according to supplied value
16     ...
       ;; set slot according to :initform
23     ...))
```

In der zweiten Phase der MCS-Lösung wird Schritt 22 durch 22a ersetzt. Die Prüfung, ob ein Slot schon gesetzt ist, entfällt. Stattdessen muß an dieser Stelle der evtl. angegebene Initialisierungswert aus der Liste der Initialisierungsargumente geholt werden. Insgesamt erhält man hier als Iterationsschritte die Anzahl der Initialisierungsargumente plus die Anzahl der Slots multipliziert mit der Anzahl der Initialisierungsargumente.

Dies bedeutet die Ersparnis einer Potenz im Polynom der Zeitkomplexität für MCS 1.0. Die Speicherkomplexität ist in MCS 1.0 günstiger, weil keine temporären Listen erzeugt werden müssen.

7.2.4 Vereinfachung der Slotzugriffe

Auch bei der Vereinfachung der Slotzugriffe wird die Kompatibilität zu CLOS auf der Objektebene beibehalten. Änderungen werden auf der Metaobjektebene vorgenommen. Statt der generischen Funktionen `slot-value-using-class` und `(setf slot-value-using-class)`, die zur Slotzugriffszeit ausgeführt werden müssen, stellt MCS 1.0 die generischen Funktionen `make-reader` und `make-writer` zur Verfügung. Sie berechnen die generischen Lese- und Schreiboperationen und können daher zur Klassenerzeugungs- und somit in der Regel schon zur Programmladezeit ausgeführt werden.

```
(make-reader class (find-slot class) 'x) 'read-slot-x)

(read-slot-x object)
  (primitive-indexed-read object 'position)

(slot-value object 'x)
  (primitive-indexed-read object
                          (find-slot-position (class-of object) 'x))
```

Nur `make-reader` und `make-writer` sind spezialisierbar. Das Konstrukt `slot-value` wird nur aus Kompatibilitätsgründen mit CLOS unterstützt. Die Implementierungsprimitive wie `primitive-indexed-read` gehören nicht zum Metaobjektprotokoll. Die berechneten Lese- und Schreiboperationen, wie z. B. `read-slot-x`, greifen in den jeweiligen Methoden differenziert auf die Slots zu. Für einfach zu ererbende Klassen, also Basisklassen, kann ein Zugriff mit bekanntem Index erfolgen. Für multipel zu ererbende Klassen, also z. B. Mixin-Klassen, wird der Index zur Zugriffszeit bestimmt. Sonst müßte man für jede erbende Klasse ggf. neue Methoden generieren, wie in MCS 0.5. Dort hing der Methoden-Lookup nicht von der Anzahl der Methoden ab, weil eine Hashtabelle für alle Methoden einer Klasse verwendet wurde. Hier werden Methoden in generischen Funktion zusammengefaßt. Für den generischen Dispatch ist es günstig, wenn eine generische Funktion genau eine Methode besitzt, weil dann nur ihre Anwendbarkeit geprüft werden muß. Der langsamere primitive Zugriff wird daher in Kauf genommen, um den Dispatch nicht zu verlangsamen. Da MCS 1.0 bereits im Kern die Mixin-Vererbung bereitstellt und nutzt, kann der langsamere primitive Slotzugriff auf Mixin-Klassen beschränkt werden. Basisklassen profitieren von beiden Optimierungen: vom schnelleren Dispatch bei nur einer Methode als auch vom schnelleren primitiven Zugriff über einen bekannten Index.

7.2.5 Mixin-Vererbung im Objektsystem-Kern

Das Konzept der Mixin-Vererbung wurde in Kapitel 4.3.3 auf Seite 93 ausführlich diskutiert. Hier beschreibe ich, wie sie in MCS 1.0 als Standardmechanismus bereits im Kern unterstützt wird. Auf der Metaobjektebene wird er verwendet, um verschiedene Verhaltensaspekte von Klassen zu klassifizieren und durch entsprechende Mixin-Klassen zu generalisieren. Der Metaobjektprogrammierer kann sie für spezielle Zwecke spezialisieren und an konkrete Anforderungen anpassen. In diesem Abschnitt gehe ich auf ihre Verwendung im Systemkern ein.

Im Designkontext von MCS 1.0 gibt es vier Verhaltensaspekte von Klassen, die man differenziert betrachten und modular implementieren möchte:

- Man will zwischen Klassen, die den Lisp-Basistypen entsprechen, und den system- bzw. benutzerdefinierten Klassen unterscheiden. Klassen, die den Lisp-Basistypen entsprechen, haben ein a priori Verhalten, das nicht aus ihrer Definition ableitbar ist. Es dürfen auch keine weiteren primitiven Klassen hinzugefügt werden. Also werden zwei Mixin-Klassen `defined` und `built-in` definiert, so daß Metaklassen entweder von der einen oder von der anderen erben dürfen.

- Klassen können abstrakt, d. h. sie besitzen keine direkten Instanzen, oder instanzierbar sein. Dies wird durch die Mixin-Klassen `abstract` und `instantiable` generalisiert.

- Aus Systemimplementierungssicht will man unterscheiden, ob Klassen einfach oder multipel ererbt werden dürfen. Davon hängt z. B. ab, ob man optimierte Lese- und Schreibmethoden generieren darf oder nicht. Entsprechend gibt es die Mixin-Klassen `single-inherited` und `multiple-inherited`.

- Das Redefinieren von Klassen wird als zusätzlicher Aspekt betrachtet, der in Erweiterungen des Objektsystemkerns eine Rolle spielt. Systemklassen sind a priori nicht redefinierbar, ohne daß es dafür eine entsprechende Mixin-Klasse gibt. In der Systemerweiterung werden die Mixin-Klassen `redefinable` und `instantiable-redefinable` definiert, auf die ich in Abschnitt 7.2.6 eingehen werde.

Abbildung 7.1 zeigt die Klassifikation der Metaklassen mit Hilfe der Mixins im Überblick. Die grundlegenden Eigenschaften aller Klassenobjekten werden in der abstrakten Basisklasse `class` definiert. Die zusätzlichen Aspekte sind in den jeweiligen Mixin-Klassen zusammengefaßt. Dabei wird deutlich, daß die Klassen eines Differenzierungspaars jeweils disjunkt sind. Es kann z. B. keine Klasse geben, die gleichzeitig abstrakt und intanziierbar ist. Das *Differenzieren* von Klassen wird bisher in keiner Programmiersprache explizit unterstützt, auch nicht in MCS. Es kann nur durch entsprechende Dokumentation der intendierten Verwendung und Spezialisierung von Mixin-Klassen zum Ausdruck gebracht werden.

Entsprechend den drei Konstrukten der Objektebene `defclass`, `defmixin` und `defabstract` gibt es die instanziierbaren Metaklassen `base-class`, `mixin-class` und `abstract-base-class` auf der Metaobjektebene. Dies sind die Default-Metaklassen, falls

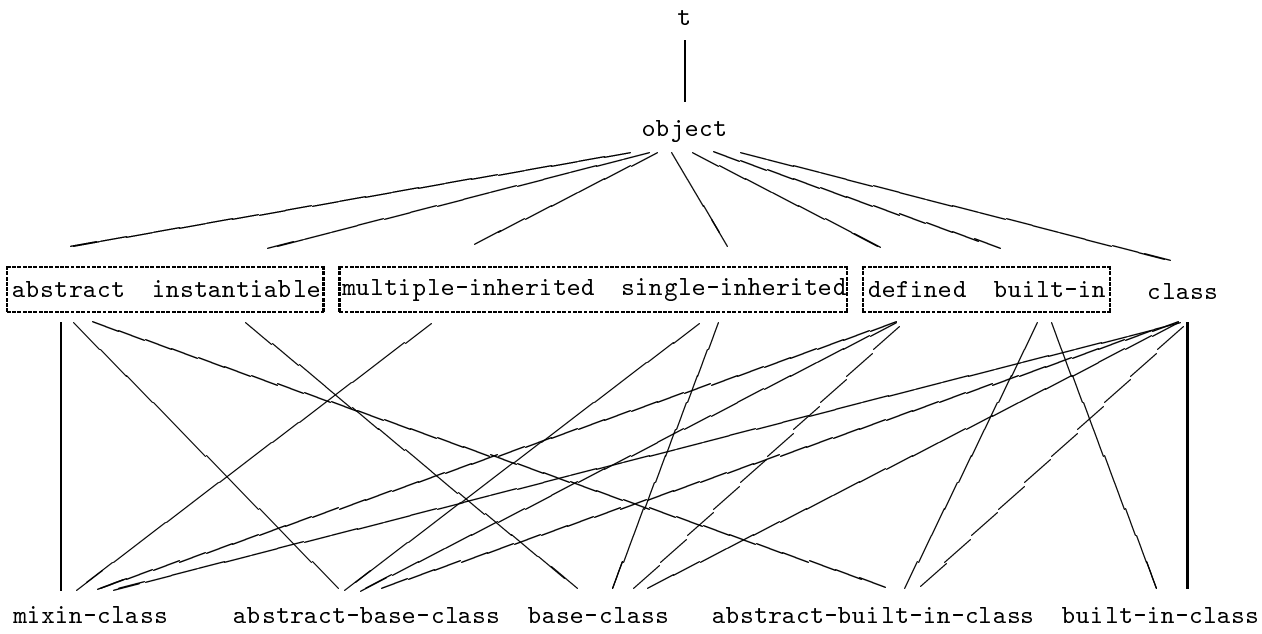


Abbildung 7.1: Klassifikation der Metaklassen mit Hilfe der Mixin-Vererbung im Überblick.

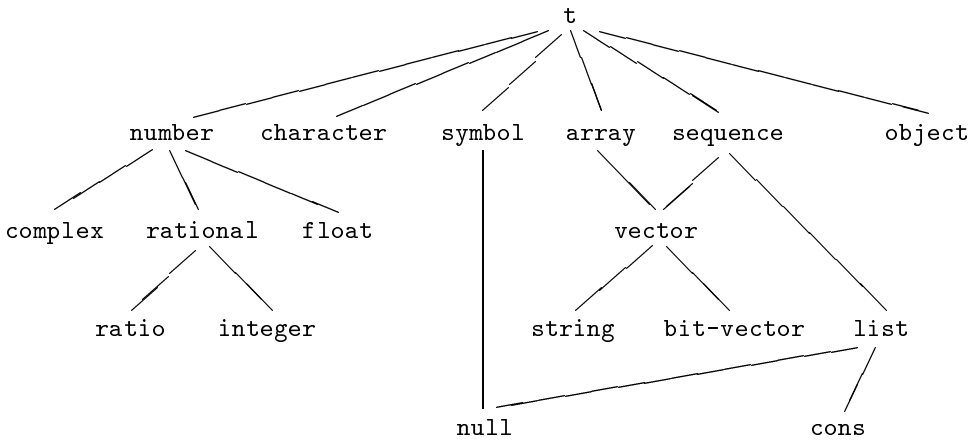


Abbildung 7.2: Klassenhierarchie der primitiven Klassen in MCS.

die Klassenoption `:metaclass` in den eben genannten syntaktischen Konstrukten nicht spezifiziert wird und falls der globale Schalter `redefine-mode` nicht aktiviert wurde. Sie können vom Benutzer weiter spezialisiert werden.

In den folgenden Unterabschnitten werden die Differenzierungsaspekte genauer dargestellt, um auch den Aufwand für die jeweilige Funktionalität zu verdeutlichen. Dies drückt sich durch zusätzlich benötigte Slots und Methoden als Teil des Struktur- und Verhaltensprotokolls aus.

Primitive vs. definierte Klassen

Gemäß der Spezifikation von CLOS werden die primitive Klassen vom Objektsystem fest vorgegeben. Primitive Klassen erben nur von `t` und nicht von `object`. Alle definierten Klassen erben von `object`. Auch die Superklassenbeziehungen primitiver Klassen sind festgelegt, siehe Abbildung 7.2. Es können keine weiteren primitiven Klassen vom Benutzer definiert werden.

Für primitive Klassen sind weder Slots noch Initialisierungsschlüsselwörter spezifiziert worden. Dennoch kann das Verhalten der Instanzen primitiver Klassen erweitert werden, indem neue generische Funktionen und Methoden definiert werden, deren Parameter über primitive Klassen diskriminieren. Diese müssen daher der Spezifikation entsprechend berechnete Klassenpräzedenzlisten besitzen. Die Unterschiede zwischen primitiven und definierten Klassen werden in Abbildung 7.3 visualisiert. Sie zeigt die abstrakte Basisklasse `class` sowie die Mixin-klassen `built-in` und `defined` als Subklassen von `object` bzw. `t`, der Wurzel der Vererbungshierarchie.

Alle hier beschriebenen Mixin-Klassen sind auf die Klasse `class` bezogen. Eine Klasse, die sie erbt, muß auch die Klasse `class` erben. Genauer gesagt, müßte sie das gleiche Protokoll erben oder selbst bereitstellen, wie es für `class` definiert ist. Dazu gehören die

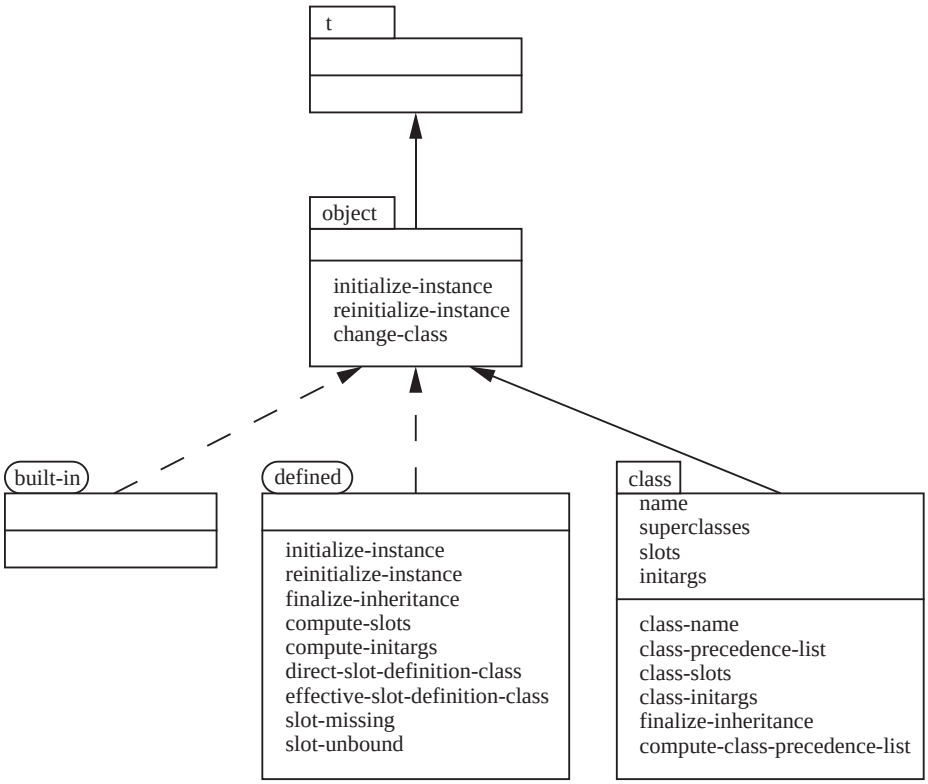


Abbildung 7.3: Differenzierung zwischen primitiven und definierten Klassen.

vier Slots `name`, `superclasses`, `slots` und `initargs`. Konzeptionell würde es ausreichen, wenn primitive Klassenobjekte nur die Slots `name` und `superclasses` besitzen würden. Aus Effizienzgründen werden jedoch auch die Slots `slots` und `initargs` allen Klassenobjekten verordnet, so daß bei primitiven Klassen ihr Wert die leere Liste ist.

Das minimale Verhalten aller Klassen ist mit den generischen Leseoperationen `class-name`, `class-precedence-list`, `class-slots` und `class-initargs` sowie den spezialisierbaren Operationen `finalize-inheritance` und `compute-class-precedence-list` spezifiziert. Die Abbildung zeigt für die jeweilige Klasse, welche Slots (oberer Teil des Kastens) und welche Methoden (unterer Teil des Kastens) für sie definiert sind. Man sieht, daß alle Aspekte von Slots und Initialisierungsschlüsselwörtern in der Mixin-Klasse `defined` konzentriert sind, bis auf o. g. effizienzbedingte Ausnahmen. Die Mixin-Klasse `built-in` enthält keine Slots und keine Methoden. Sie hat somit einen rein prädikativen Charakter.

Abstrakte vs. instanziierbare Klassen

Die Unterschiede zwischen abstrakten und instanziierbaren Klassen werden in Abbildung 7.4 gezeigt. Die Klassen `class`, `object` und `t` wiederholen sich unverändert. Die Mixin-Klassen `abstract` und `instantiable` beschreiben einen weiteren Klassenaspekt, nämlich die Instanziierbarkeit.

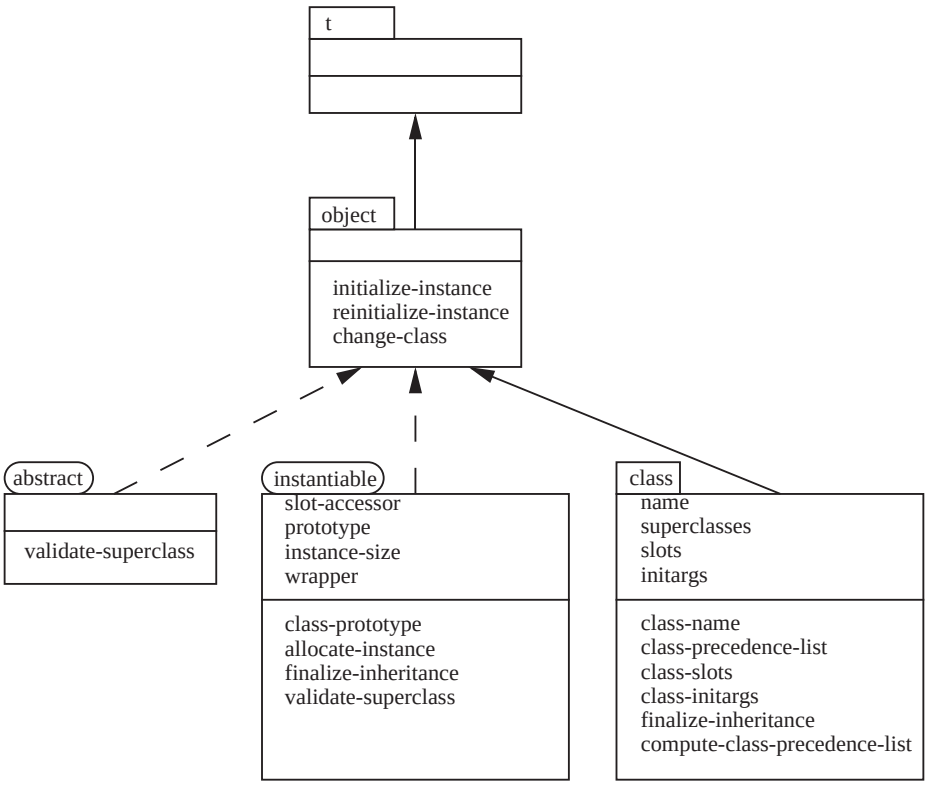


Abbildung 7.4: Differenzierung zwischen abstrakten und instanziierbaren Klassen.

Gegenüber abstrakten Klassen benötigen die instanziierbaren doppelt so viele Slots. Geht man davon aus, daß die Hälfte aller Klassen eines komplexen Softwaresystems abstrakt

sind (Mixin-Klassen und abstrakte Basisklassen), so wird mit MCS der Speicherbedarf für alle Klassenobjekte um ein Viertel reduziert werden.

Einfach vs. multipel zu ererbende Klassen

Die Unterschiede zwischen einfacher und multipler Vererbung wurden schon ausführlich diskutiert. Abbildung 7.5 zeigt, wie sie sich in MCS 1.0 konkret auf die zu definierenden Methoden der Metaobjekteben auswirken. Zum einen betrifft dies die Überprüfung der Vererbungs-Zulässigkeit mit `validate-superclass`, zum anderen die Berechnung der Lese- und Schreiboperationen mit `make-reader` und `make-writer`.

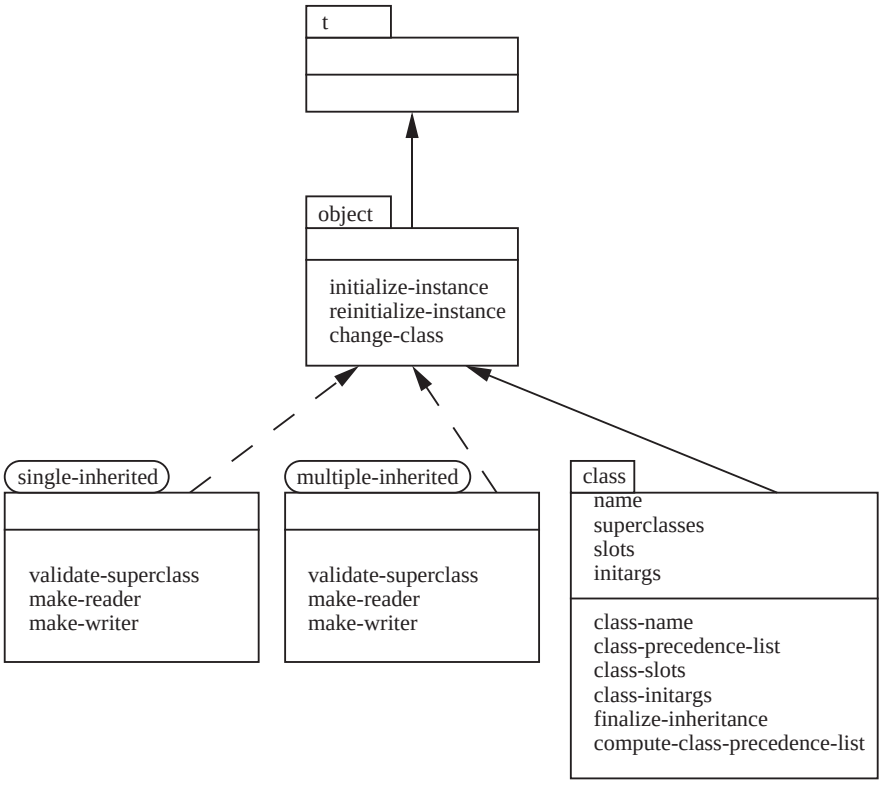


Abbildung 7.5: Differenzierung zwischen einfacher und multipler Vererbungsstrategie.

Die Anwendung der Mixin-Vererbung auf den Objektsystemkern selbst, hat signifikant dazu beigetragen, die verschiedenen Aspekte von Klassenobjekten zu explizieren, sie gezielt einzusetzen und effizient zu implementieren. Ein Nachteil ist nur das kompliziertere Bootstrapping des Objektsystemkerns.

7.2.6 Redefinieren und Reklassifizieren als Erweiterung

Das Redefinieren von Klassen mit automatische Anpassung betroffener Instanzen sowie das Reklassifizieren von Instanzen (`change-class`) wurde aus zwei Gründen als Erweiterung des Objektsystemkerns realisiert. Zum einen gibt es für diese Funktionalität einen

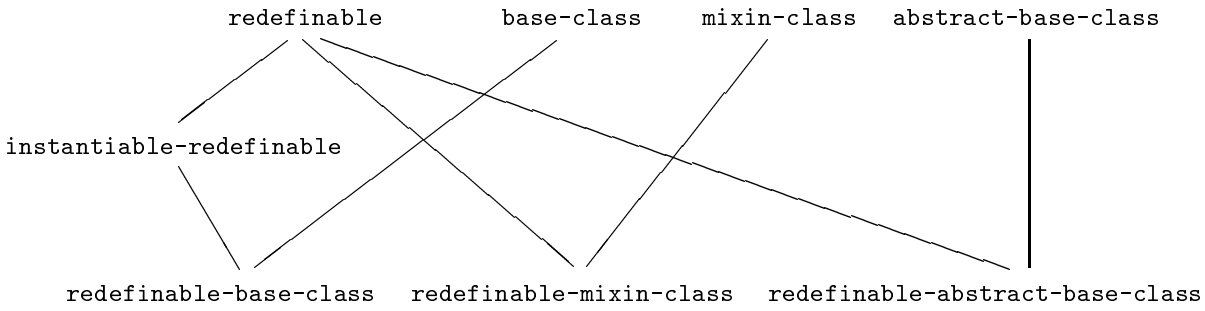


Abbildung 7.6: Klassifikation redefinierbarer Klassen mit Hilfe der Mixin-Vererbung im Überblick.

berechtigten Bedarf zur Programmentwicklungszeit. Zum anderen kann so der Mehraufwand für diese Funktionalität im Vergleich zum Standardfall expliziert werden. Den höheren Preis für die erweiterte Funktionalität müssen nur solche Programme zahlen, die sie auch wirklich nutzen. Abbildung 7.6 zeigt die Klassenhierarchie redefinierbarer Klassen im Überblick. Hier wurden die Superklassen der Metaklassen nicht `base-class`, `mixin-class` und `abstract-base-class` nicht wiederholt. Die Mixin-Klasse `redefinable` ist Subklasse von `object`.

Abbildung 7.7 zeigt einen Teil der Klassenhierarchie redefinierbarer Klassen mit den zusätzlichen Slots und Methoden. Dabei wurde aus Platzmangel nur die Klasse `redefinable-abstract-base-class` weggelassen.

Neben den zwei Mixin-Klassen `redefinable` und `instantiable-redefinable` werden in dieser Erweiterung die instanziierbaren Metaklassen `redefinable-base-class`, `redefinable-mixin-class` und `redefinable-abstract-base-class` bereitgestellt. Dies sind die Default-Metaklassen, falls die Klassenoption `:metaclass` in den syntaktischen Konstrukten `defclass`, `defmixin` und `defabstract` nicht spezifiziert und der globale Schalter `redefine-mode` aktiviert wurde. Bei der Definition von Klassen hat der Benutzer somit die freie Wahl, die Klassenart über die Option `:metaclass` explizit zu bestimmen oder implizit vom Zustands des Schalters `redefine-mode` abhängig zu machen. Letzterer beeinflußt nur das Verhalten der Konstrukte `defclass`, `defmixin` und `defabstract`, nicht der Klassenobjekte an sich. Sowohl zur Entwicklungs- als auch zur Ausführungszeit kann es in einem System redefinierbare und nicht-redefinierbare Klassen geben.

Um nicht unnötig viele Metaklassen definieren zu müssen, wird der Aspekt des Reklassifizierens von Objekten mit redefinierbaren Klassen verknüpft. Der Implementierungsaufwand für das Redefinieren deckt die Anforderungen des Reklassifizierens im wesentlichen mit ab. Hinzu kommt die Methoden `change-class-using-class` in der Mixin-Klasse `instantiable-redefinable`.

Ohne auf weitere Klassen des Kerns von MCS 1.0 genauer einzugehen, sind sie im Überblick in Abbildung 7.8 dargestellt.

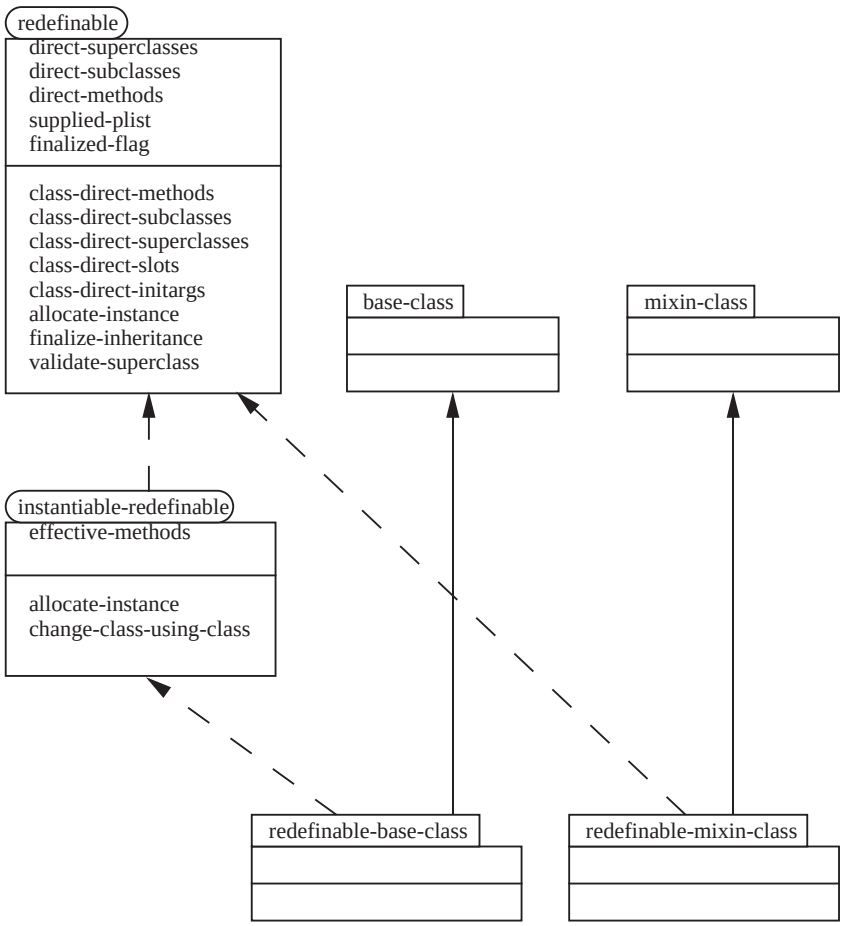


Abbildung 7.7: Klassenhierarchie redefinierbarer Klassen.

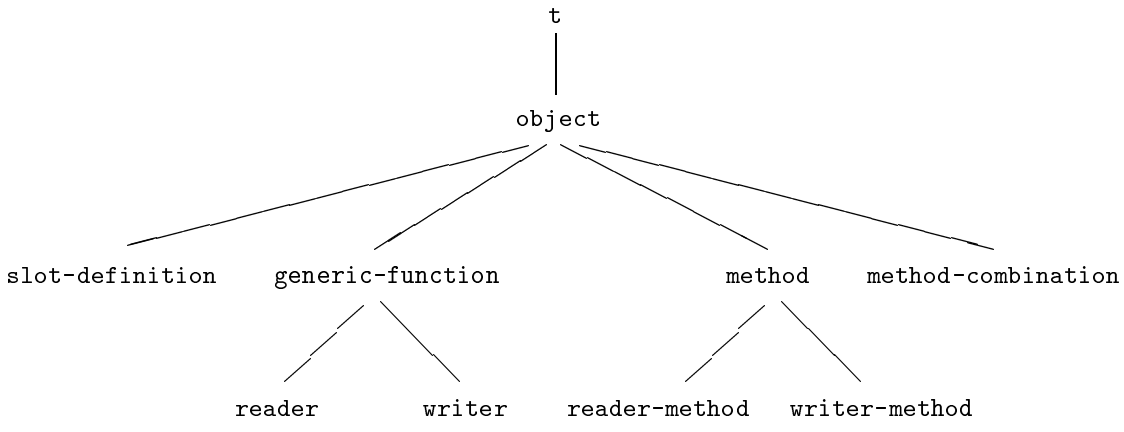


Abbildung 7.8: Weitere Metaobjektklassen in MCS 1.0.

Im nächsten Abschnitt wird nun die Performanz von MCS 1.0 diskutiert.

7.3 Performanz von MCS 1.0

Das Ziel bei der Implementierung von MCS 1.0 war, die Effizienz von MCS 0.5 beizubehalten und die Schwächen beim Nachrichtenversenden im Vergleich zu PCL bzw. zum kommerziellen Systems Lucid Common Lisp FLAVORS zu beseitigen. Es wurden zahlreiche Performanzmessungen durchgeführt, deren wesentliche Ergebnisse hier zusammengefaßt werden. Die durchgeführten Tests sind ähnlich den Tests mit MCS 0.5, sie sind aber auf die Sprachkonstrukte von CLOS ausgerichtet.

Der letzte Abschnitt hat gezeigt, welche Speicherplatzeinsparung die gezielte Verwendung der Mixintechnik bewirken kann. In CLOS werden alle Klassen einheitlich auf den größten gemeinsamen Nenner gebracht: alle Klasse sind potentiell instanziierbar und redefinierbar. Vergleicht man den Speicherbedarf von CLOS-Klassen mit dem minimalen Fall der Mixin- bzw. der abstrakten Basisklassen in MCS, so ist das Verhältnis 7 zu 2. Daher verwundert es nicht mehr, daß MCS 1.0 bzgl. des Speicherbedarfs wesentlich besser abschneidet als PCL oder CLOS in Allegro COMMONLISP, und zwar aus prinzipiellen Gründen.

Bevor ich auf die Einzelheiten der Laufzeit-Tests eingehe, rufe ich nochmal den Anwendungskontext in Erinnerung. MCS 1.0 wurde als objektorientierte Implementierungsgrundlage für die Entwicklung des kommerziellen Produkts babylon aus dem Forschungsprototypen BABYLON ausgewählt, weil schon bei den ersten Vergleichsmessungen MCS am besten abschnitt. Die Anforderungen der Anwendung babylon haben den Zuschnitt der folgenden Tests insofern beeinflußt, als die Gewichtung der Einzeltests zueinander entsprechend gewählt wurde. Diese Anforderungen sind aber generell genug, um ein breites Spektrum komplexer Softwaresysteme abzudecken. Die folgende Tabelle 7.9 zeigt, wie sich der Umfang des Quellcodes gegenüber dem Forschungsprototypen veränderte.

Parameter	BABYLON	babylon	Veränderungsfaktor
Klassen	149	395	2.65
Operationen	596	2450	4.11
Methoden	899	3635	4.04
Quellcode in MB	2	8	4

Abbildung 7.9: Umfang des Quellcodes von babylon in seiner Entwicklung.

Der Gesamtumfang des Quellcodes hat sich vervierfacht. Lediglich die Anzahl der Klassen ist nur um den Faktor 2,65 gestiegen, währenden sich die Anzahl der Operationen und Methoden ebenfalls vervierfacht hat, worauf auch der Gesamtanstieg zurückzuführen ist. Das Verhältnis von Klassen zu Operationen bzw. zu Methoden hat sich somit verschoben, und zwar von $1:4:6$ auf $1:6:9$. Interessanterweise hat sich die durchschnittliche Anzahl von Methoden pro generische Operation nicht verändert, sie liegt bei 1,5.

In den durchgeführten Laufzeit-Tests bin ich also von einem gerundeten Verhältnis von Klassen zu Methoden eines realen Systems ausgegangen:

- **t1** erzeugt mit `defclass` zur Laufzeit 10 Klassen.
- **t2** erzeugt mit `defmethod` zur Laufzeit 100 Methoden.
- **t3** erzeugt mit `make-instance` zur Laufzeit 1000 terminale Instanzen von Klassen.
- **t4** greift mit `slot-value` 10000 mal auf Slots zu.
- **t5** führt 10000 mal eine generische Leseoperation durch.
- **t6** führt 10000 mal eine generische Schreiboperation durch.
- **t7** führt 10000 mal den generischen Dispatch über *ein* Argument durch.
- **t8** führt 10000 mal den generischen Dispatch über *zwei* Argumente durch.
- **t5a-d** führen 10000 mal eine generische Leseoperation jeweils mit einer Primär-, zusätzlich mit einer Before-, zusätzlich mit einer After und schließlich auch einer Aroundmethode durch. Hiermit wird die Effizienz der Methodenkombination getestet.

Die Meßergebnisse für PCL und MCS in Allegro Common Lisp 3 sowie für *native* CLOS und MCS in Allegro Common Lisp 4.0 werden hier in vier Diagrammen graphisch dargestellt. Im Anhang C.1 findet man die genauen Zahlen.

Abbildung 7.10 zeigt die Ausführungszeiten der Klassen- und Methodendefinition sowie der Instanzerzeugung (Tests **t1**, **t2** und **t3**). Normalerweise betreffen die beiden ersten Tests die Ladezeit einer Anwendung und der dritte die eigentliche Laufzeit. Im Fall von babylon als einer Entwicklungs- und Ablaufumgebung für wissensbasierte Systeme ist jedoch auch die Laufzeit für Klassen- und Methodendefinitionen kritisch, weil sie die Entwicklungszeit wissenbasierter Endanwendungen direkt beeinflussen.

Wie man der Abbildung 7.10 entnehmen kann, schneidet MCS dank den getroffenen Designentscheidungen hier überragend besser ab. Gegenüber PCL ergibt sich ein Faktor 7 bei der Klassendefinition und ein Faktor 11 bei der Methodendefinition. Selbst gegenüber dem kommerziellen CLOS in Allegro CommonLisp beträgt der Abstand einen Faktor 4 bei der Klassen- bzw. 5 bei der Methodendefinition. Dabei ist hier anzumerken, daß auch in MCS redefinierbare und instanziierbare Klassen erzeugt wurden, also der komplexeste Fall gemessen wurde. Alle anderen Klassen, die ja den Standardfall ausmachen, werden noch schneller erzeugt und initialisiert, insbesondere die Mixin-Klassen. Wie im Abschnitt 7.2.5 zu sehen war, brauchen für sie nur wenige Methoden ausgeführt zu werden.

Die Instanzerzeugung (**t3**) betrifft alle Phasen der Nutzung von babylon, sowohl der Entwicklung wissensbasierter Systeme als auch deren Ablauf als Endanwendung. Auch hier zeigt sich ein deutlicher Abstand: Faktor 9 zu PCL und Faktor 2 zu CLOS.

Abbildung 7.11 zeigt die Ausführungszeiten der Slotzugriffe über `slot-value` (**t4**) und über generische Lese- und Schreibfunktionen (**t5** und **t6**). Neben der Instanzerzeugung ist es die Effizienz der Slotzugriffe, die die Akzeptanz objektorientierter Sprachen begünstigt oder erschwert. Hier fallen die Abstände geringer aus und im Fall generischer Lesezugriffe

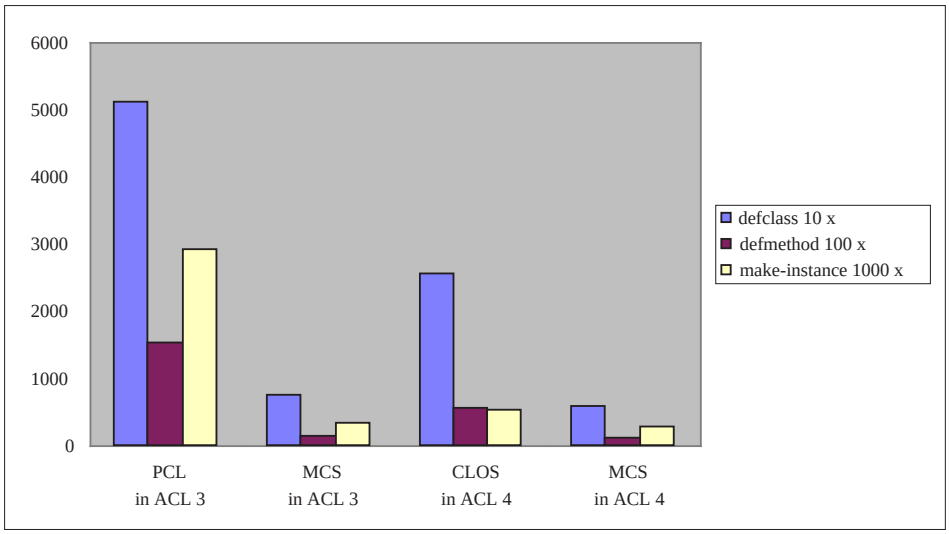


Abbildung 7.10: Ausführungszeiten der Klassen- (**t1**) und Methodendefinition (**t2**) sowie der Instanzerzeugung (**t3**) in PCL, Franz Allegro CLOS und MCS 1.0 in Millisekunden.

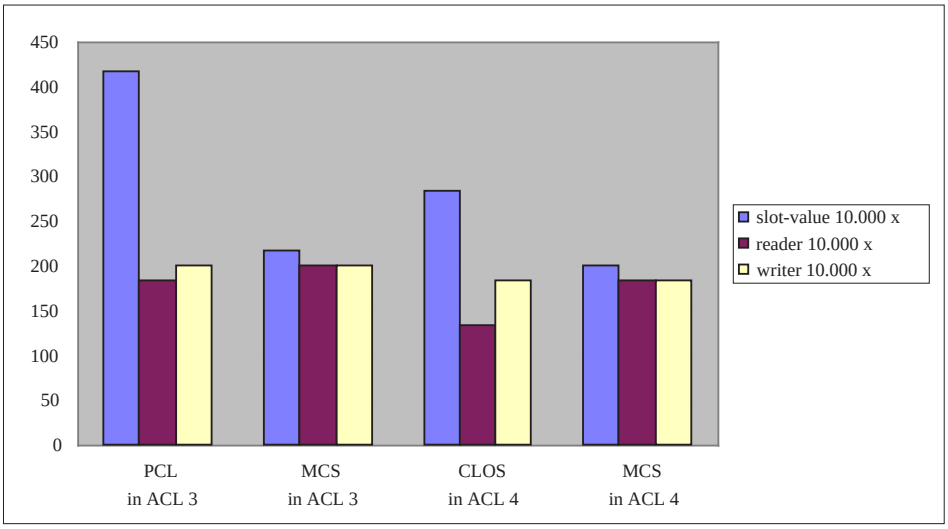


Abbildung 7.11: Ausführungszeiten der Slotzugriffe (**t4**, **t5** und **t6**) in PCL, Franz Allegro CLOS und MCS 1.0 in Millisekunden.

ist MCS sogar etwas langsamer als PCL bzw. CLOS. Der Grund hierfür liegt in aufwendigen implementierungsabhängigen Optimierungen in PCL und weiteren Optimierungen auf Maschinenbefehlsebene in CLOS. In MCS wurden nur portable Sprachmittel von CommonLisp verwendet.

Abbildung 7.12 zeigt die Ausführungszeiten des generischen Dispatchs bei Diskriminierung über *ein* (**t7**) bzw. über *zwei* Argumente (**t8**). Der Test **t7** zeigt zufällig das gleiche

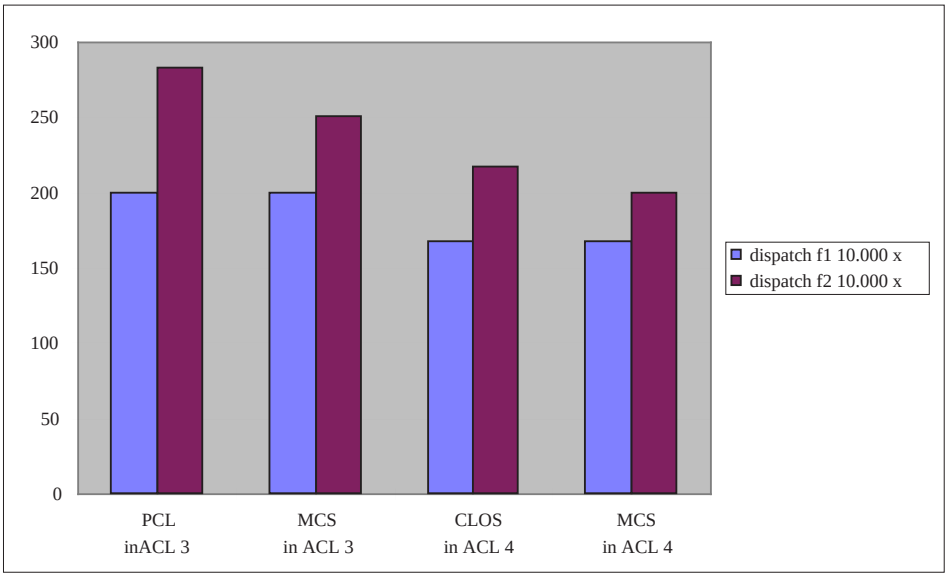


Abbildung 7.12: Ausführungszeiten des generischen Dispatchs (**t7** und **t8**) in PCL, Franz Allegro CLOS und MCS 1.0 in Millisekunden.

Meßergebnis jeweils für PCL und MCS in Allegro Common Lisp 3.0 bzw. MCS und CLOS in Allegro Common Lisp 4.0, obwohl die Implementierungstechnikten unterschiedlich sind. Aber bereits bei zweifachem Dispatch greift das vereinfachte Design von MCS stärker als aufwendige Optimierungen auf maschinennäheren Implementierungsebenen bei PCL und CLOS.

Abbildung 7.13 zeigt das Performanzverhalten bzgl. der Methodenkombination. Es werden die Ausführungszeiten einer Leseoperation mit nur einer Primärmethode (**t5a**), einer Primär- und einer Before-Methode (**t5b**), einer Primär-, einer Before- und einer After-Methode (**t5c**) sowie mit einer Primär-, einer Before-, einer After-Methode und einer Around-Methode (**t5d**) verglichen. Die zur Primärmethode hinzukommende Ausführungszeit ist reiner *Overhead* der Methodenkombination. Die Sekundärmethoden tun nichts, sie enthalten lediglich eine Pseudoanweisung.

Hier zeigt sich ein gewisses Sprungverhalten bei PCL und CLOS. Der Spezialfall einer Leseoperation mit nur einer Primärmethode ist besonder optimiert. Durch Hinzukommen einer Before-Methode werden die Optimierungen hinfällig, die Ausführungszeiten steigen um den Faktor 7 bzw. 4. Eine weitere After-Methode erhöht die Ausführungszeit nur

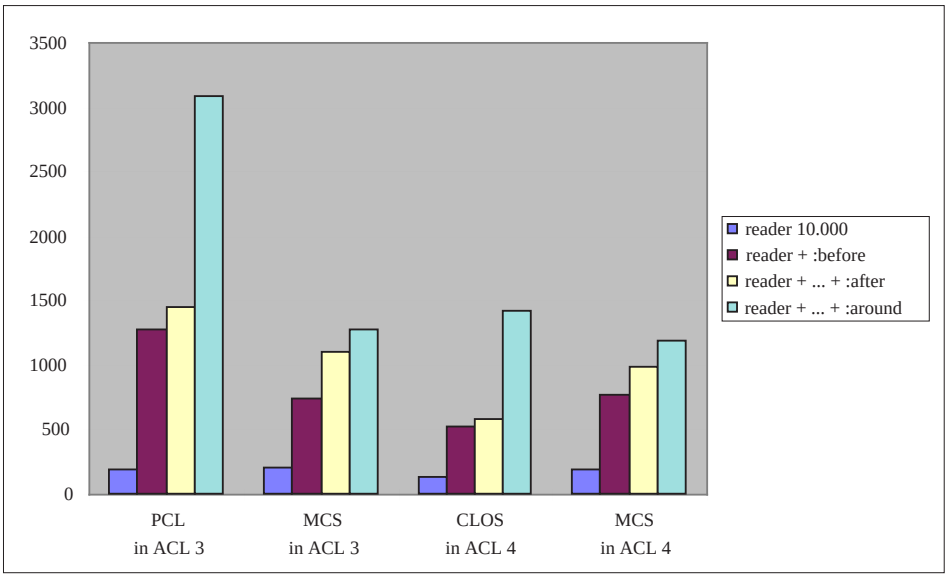


Abbildung 7.13: Ausführungszeiten der Methodenkombination (**t5a**, **t5b**, **t5c** und **t5d**) in PCL, Franz Allegro CLOS und MCS 1.0 in Millisekunden.

geringfügig. Einen zweiten Sprung bewirkt die Around-Methode: Faktor 2,1 bei PCL bzw. Faktor 2,4 bei CLOS. Dieser zweite Sprung bleibt bei MCS aus. Dank der Vereinfachung des Konstrukts `call-next-method` gewinnt MCS hier einen Abstand um den Faktor 2,4 gegenüber PCL und um den Faktor 1,2 gegenüber CLOS. Dieses Ergebnis ist ein Grund dafür, in ΤΕΛΟΣ nur noch Primärmethoden mit dem Konstrukt `call-next-method` zu unterstützten und auf Sekundärmethoden ganz zu verzichten.

Und hier folgen die Summen der Tests:

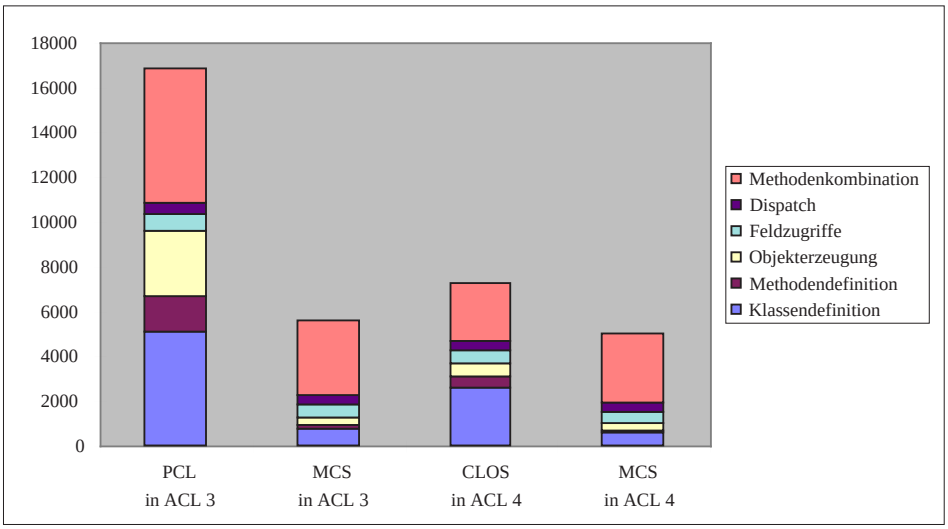


Abbildung 7.14: Summe der Ausführungszeiten aller Tests in PCL, Franz Allegro CLOS und MCS 1.0 in Millisekunden.

7.4 Bewertung

Im Vergleich mit MCS 0.5 wurde mit MCS 1.0 ein großer Schritt von einem kleinen Objektsystem nach dem Modell des Nachrichtenaustauschs und der allgemeinen multiplen Vererbung hin zu Konzepten generischer Funktionen mit mehrfachem Dispatch und der Mixin-Vererbung gemacht. Unter Beibehaltung der metareflektiven Systemarchitektur und der wesentlichen Implementierungstechniken wurde ein neues Objektsystem realisiert, das auf der Objektebene voll CLOS-kompatibel ist und das auf der Metaobjektebene im Vergleich zu CLOS MOP bereits eine neue Richtung einschlägt. Die Grundkonzepte wurden dabei von CLOS übernommen. Ihre konkrete Gestaltung wurde aber nach den Kriterien aus Kapitel 4 vorgenommen, was zu ihrer Vereinfachung und somit zu wesentlichen Effizienzsteigerungen, z. B. gegenüber PCL und Franz Allegro CLOS, geführt hat.

Mit MCS 1.0 wird erstmals das Konzept der Mixin-Vererbung explizit unterstützt. Im Objektsystemkern selbst wird diese Klassifikationstechnik verwendet, um verschiedene Aspekte von Klassenobjekten bzgl. der Funktionalität aufzuzeigen und bzgl. der spezialisierbaren Implementierung den jeweiligen Aufwand abzugrenzen. Es hat sich erwiesen, daß

das Redefinieren von Klassen und Reklassifizieren von Instanzen einen signifikant höheren Speicherbedarf bei der Repräsentation von Klassen und Instanzen nach sich zieht. Mixin-Klassen, als nicht-redefinierbare und nicht-instanziierbare Klassen bieten weitere Möglichkeiten, den Speicherbedarf gegenüber instanziierbaren Basisklassen zu halbieren. CLOS hingegen erlaubt zwar dem Benutzer, Klassen nach dem Mixin-Konzept zu organisieren, unterstützt dies aber nicht und zieht auch selbst keinen Nutzen daraus.

Einziger Nachteil der ausgesprochen objektorientierten Realisierung des Objektsystemkerns selbst ist das kompliziertere Bootstrapping durch die höhere Zahl beteiligter Metaobjektklassen. In CLOS ist sie aus anderen Gründen auch nicht niedriger.

Das Konzept generischer Funktionen aus CLOS verbessert die Integration objektorientierter Sprachmittel mit den funktionalen Konstrukten von Lisp im Vergleich zu MCS 0.5 bzw. zu FLAVORS. MCS 1.0 vereinfacht generische Funktionen bzgl. der Parameterlisten. Es verzichtet auf multiple Ergebniswerte und auf instanzspezifische Methoden. Dadurch ist die portable Implementierung des generischen Dispatchs genauso effizient, wie die aufwendige Lösung mit Optimierungen auf tieferen Implementierungsebenen in PCL und CLOS. Durch den Verzicht auf optionale Argumente bei `call-next-method` ist der portable Dispatch in MCS bei vorhandenen Sekundärmethoden effizienter als in PCL und CLOS. Bei Lese- und Schreiboperationen bleibt der portable Dispatch etwas langsamer.

Die Vereinfachungen bei der Instanzerzeugung und bei Slotzugriffen haben sowohl für ein klareres Konzept als auch für eine wesentlich effizientere Implementierung als in PCL bzw. CLOS gesorgt. Quellcode-Interpretation wurde in der Implementierung von MCS 1.0 von vornherein ausgeschlossen, und die inkrementelle Kompilation wurde schrittweise reduziert. Dabei gelang es erstmals, eine spezielle Version von MCS 1.0 mit dem Komplett-Compiler CLiCC zu übersetzen.

Die Metaobjektebene von MCS 1.0 unterscheidet sich schon deutlich von CLOS MOP. Das Kompatibilitätskriterium hatte hier ein geringeres Gewicht als auf der Objektebene. Eine wesentliche Rolle spielen hier die Mixin-Metaklassen mit den jeweiligen Slots und Methoden. Das Metaobjektprotokoll von MCS 1.0 ist deutlich mächtiger und besser spezialisierbar als in MCS 0.5. Es unterscheidet sich von CLOS MOP dadurch, daß es funktionaler ist. Die Eingangsparameter für Berechnungen werden auch formal als Argumente durchgereicht, sie müssen nicht durch Aufruf von Leseoperationen aus möglicherweise nocht nicht vollständig initialisierten Klassenobjekten ermittelt werden.

Dennoch ist MCS 1.0 noch zu stark an CLOS orientiert, was sich z. B. an generischen Lese- und Schreiboperationen für alle Klassen zeigt. Das Slotzugriffsprotokoll ist noch zu grobkörnig und erzwingt explizite Operationen auf der globalen Bindungsumgebung in den generischen Funktionen `make-reader` und `make-writer`, was Probleme mit der Komplettkompilation bereitet. Ein weiterer Nachteil ergibt sich durch das Fehlen eines Modulkonzepts. Die Packages von COMMONLISP helfen zwar, einige Probleme, wie Namenskonflikte, zu vermeiden, sie schaffen aber keinen syntaktische Abschluß, um sichere globale Optimierungen vorzunehmen.

Bedenkt man, daß die von außen herangetragene Anforderung weitgehender Kompatibilität mit CLOS die Möglichkeit schuf, MCS 1.0 im Produktentwicklungsprojekt babylon unter denkbar harten Randbedingungen einzusetzen, so hat es sich doch gelohnt, die Nachteile der Kompatibilität in Kauf zu nehmen. Der reale Test hat die Konzepte und ihre Implementierung in einer Weise bestätigt, wie man es sonst nicht erwarten könnte.

Die verbliebenen Schwächen von MCS 1.0, die größtenteils mit der Kompatibilität mit CLOS verbunden sind, werden nun im nächsten Kapitel mit dem Entwurf von ΤΕΛΟΣ als Teil des neuen Lisp-Dialekts EuLisp beseitigt.

Kapitel 8

Die Reflektion: Entwurf und Implementierung von TELOS

I will remark in passing that the opportunity to do an all-new design is really wonderful.

Frederick P. Brooks, Jr. 1993. Keynote address: Language Design as Design, The Second History of Programming Languages Conference (HOPL-II), [Brooks, 1996, S. 11].

Im ersten Schritt des experimentellen Teils dieser Arbeit wurde mit MCS 0.5 ein erweiterbarer minimaler Objektsystemkern nach dem Modell des Nachrichtenaustauschs realisiert. Die metareflektive Systemarchitektur eröffnete einen Lösungsweg, das Objektsystem selbst objektorientiert zu beschreiben und zu implementieren. Durch ein gezieltes Design der Sprachkonstrukte, konnte in der Implementierung jegliche Quellcode-Interpretation zur Ausführungszeit vermieden werden. Darin liegt ein genereller Schlüssel zur Realisierung effizienter komplexer Lisp-Systeme. In der Implementierung von MCS 0.5 wurde aber intensiv der inkrementelle Compiler von CommonLisp verwendet, hauptsächlich zur Klasseninitialisierungs- und somit zur Ladezeit. Dies beeinträchtigte die Effizienz des Objektsystems zumindest in einigen CommonLisp-Systemen. Die Beschränkung des Metaobjektprotokolls auf Klassen als einzige Metaobjekte erwies sich als nachteilig für portable Spracherweiterungen. Bereits bei Feldannotationen mußte man auf die Implementierungsebene absteigen.

Mit MCS 1.0 wurde eine insgesamt deutlich ausdrucksstärkere objektorientierte Sprache als bei MCS 0.5 realisiert. Wie auch CLOS basiert sie auf dem Konzept generischer Funktionen an Stelle des Nachrichtenversendens. Im Unterschied zu CLOS wurden viele Sprachkonstrukte vereinfacht. Einige Konzepte, wie das Redefinieren von Klassen, wurden aus dem Standardverhalten herausgenommen und nur als Erweiterung des Objektsystemkerns angeboten. Neben Vereinfachungen der CLOS-Konstrukte wurde mit MCS 1.0 erstmalig das Konzept der Mixin-Vererbung explizit unterstützt. Sowohl im Design als auch in der Implementierung von MCS 1.0 selbst wird die Mixin-Vererbung zur Klassifikation der Struktur und des Verhaltens der Metaobjekte verwendet.

Die Schwächen von MCS 1.0 ergeben sich hauptsächlich aus der selbstauferlegten Kom-

patibilität mit CLOS und aus seiner Einbettung in CommonLisp. Das Fehlen eines klassischen Modulkonzepts in CommonLisp wirkt sich besonders nachteilig aus. Auch wenn die Nützlichkeit der Mixin-Vererbung außer Zweifel steht, muß doch die Frage gestellt werden, ob der Objektsystemkern dadurch nicht zu groß und das Bootstrapping nicht zu kompliziert wird.

Mit der gesammelten Erfahrung im Hintergrund und einem modernen Lisp in der Definitionsphase bietet sich nun die einmalige Chance, ein von Grund auf integriertes Objektsystem für EuLisp zu entwerfen. Die Hauptmerkmale von Τελος sind:

- ein auf einfacher Vererbung und generischen Funktionen beruhender metareflektiver Kern,
- ein modulares, semantisch robustes und einfach spezialisierbares Metaobjektprotokoll,
- Mixin-Vererbung als portable Standarderweiterung des Kerns,
- strikte Trennung zwischen der Objekt- und der Metaobjektebene dank des Modulsystems von EuLisp.

Im wesentlichen geht es in diesem Kapitel darum, die objektorientierten Konzepte aus Kapitel 4 in einem konkreten Objektsystem zu realisiert. Die grundlegenden Implementierungstechniken wurden in Kapitel 5 diskutiert und brauchen hier nicht wiederholt zu werden. Ebensowenig muß auf Dinge eingegangen werden, die in MCS 0.5 und MCS 1.0 schon ausführlich behandelt wurden und sich in Τελος nicht ändern. Herausgestellt werden hier die Optimierungsmöglichkeiten durch Modul- und Applikations-Compiler.

Da es aus naheliegenden Gründen zum Zeitpunkt des Entwurfs von Τελος noch keine ausgereifte Programmierumgebung für EuLisp gab, habe ich die Referenzimplementierung in CommonLisp mit Hilfe der Einbettungstechnik vorgenommen. Mit der Verfügbarkeit eines vollständigen EuLisp-Compilers erübrigt sich natürlich der Bedarf nach einer optimierenden portablen Implementierung von Τελος in CommonLisp oder auch in EuLisp selbst. Meine Referenzimplementierung von Τελος, die ich hier präsentiere, stellt daher auch nur eine Übergangslösung dar. Die potentielle Effizienz von Τελος kann natürlich erst mit einem vollständigen Modul- und Applikations-Compiler für EuLisp erreicht werden, denn darauf ist das Sprachdesign ausgerichtet. Die Compiler-Konstruktion selbst liegt außerhalb des Rahmens dieser Arbeit. Im Unterschied zu den beiden vorausgegangenen Systemen MCS 0.5 und MCS 1.0, beschreibe ich in diesem Kapitel, welche Compiler-Analysen und -Optimierungen auf Grund getroffener Designentscheidungen möglich werden. Aber auch die portable Referenzimplementierung Celos zeigt bessere Effizienzresultate als heutige kommerzielle CLOS-Implementierungen.

In diesem abschließenden Kapitel diskutiere ich zunächst die Basissprache EuLisp sowie die Grundzüge seines Objektsystems Τελος. Anschließend entwerfe ich eine Version von Τελος, die fast vollständig meiner Sicht objektorientierter Konzepte entspricht. In einer Vorversion 0.99 wurde das unter meiner Federführung revidierte Τελος von der EuLisp Working Group akzeptiert und in [Padget *et al.*, 1993] publiziert. Für die hier präsentierte Version 1.0 bin ich allein verantwortlich. Sie wurde in dieser Form weder diskutiert noch veröffentlicht. Teilergebnisse wurden in [Bretthauer *et al.*, 1993] und [Bretthauer *et al.*,

1994] publiziert. TELOS 1.0 geht über die Version 0.99 hinaus und löst einige Probleme, die vorher noch offen waren. Dazu zählt die Behandlung der Feldnamen und der Initialisierungsschlüsselwörter. Zum einen genügen diese nun den Anforderungen der Kapselung, indem sie den Regeln des Imports und Exports von Bindungen auf Modulebene folgen. Gleichzeitig können bzgl. ein und desselben Feldes jeweils mehrere Bezeichner für den Feldnamen und für das Initialisierungsschlüsselwort verwendet werden. Intern besitzt jedoch jedes Feld genau *einen* Namen und höchstens *ein* Initialisierungsschlüsselwort. Eine weitere Neuerung ist ein Konzept zur Erzeugung und Initialisierung von Objekten, das auf dem gleichen Prinzip wie das Feldzugriffsprotokoll beruht und in Kapitel 4 bereits diskutiert wurde. Schließlich ist die Erweiterung der Methodenkombination zu nennen, die ebenso deklarativ ist wie in CLOS, aber gleichzeitig die Designkriterien aus Kapitel 2 erfüllt.

8.1 EuLisp und TELOS im Überblick

EuLisp [Padget *et al.*, 1993] wurde als eine moderne universelle Programmiersprache mit dem Ziel entwickelt, die traditionellen Stärken von Lisp im Kern zu erhalten und die Schwächen durch bessere Konzepte aus anderen Programmiersprachen auszugleichen. Insbesondere sollte das Sprachdesign einfache und effiziente Implementierungen ermöglichen. EuLisp zeichnet sich durch drei Hauptmerkmale aus:

- Klassische Lisp-Basistypen und das Objektsystem werden in *einer* uniformen Klassenhierarchie integriert.
- Klassifikation und Modularisierung werden als komplementäre Abstraktionsmittel durch orthogonale Sprachkonzepte unterstützt.
- Nebenläufigkeit ist von vornherein integriert.

Daraus leiten sich folgende Sprachkonzepte ab:

- Das Modulsystem dient der Kapselung und separaten Kompilation syntaktisch abgeschlossener Programmeinheiten. Jedes Modul definiert eine eigene lexikalische Bindungsumgebung. Bindungen können importiert und exportiert werden. Bestimmte Eigenschaften des Objektsystems werden durch das Modulsystem erst möglich.
- EuLisp selbst ist modular strukturiert. Der Sprachkern besteht aus zwei Ebenen: die Objektebene `level-0` enthält syntaktische Konstrukte, Klassen und Funktionen; die Metaobjektebene `level-1` bietet weitere syntaktische Konstrukte, Metaobjektklassen und weitere Funktionen, insbesondere die des Metaobjektprotokolls.[1] Die volle Funktionalität der Sprache wird in Form von Erweiterungsmodulen auf der jeweiligen Ebene bereitgestellt.
- Nebenläufigkeit basiert auf einfachen Primitiven: *Threads* und Semaphoren. Abstraktere Mechanismen werden in entsprechenden Erweiterungsmodulen bereitgestellt.
- Funktionen und *Downward-Continuations* sind Objekte erster Ordnung. Sie sorgen für flexible Kontrollstrukturen.

[1] Die Metaobjektebene ist nicht notwendigerweise mit Mitteln der Objektebene implementierbar.

- Das *Condition-System* zur Ausnahmebehandlung verwendet Klassen, Threads und generische *Handler*.
- Makros ermöglichen syntaktische Erweiterungen.
- Dynamische Variablenbindungen werden syntaktisch klar von lexikalischen Bindungen unterschieden. Sie spielen ohnehin eine untergeordnete Rolle in EuLisp.
- Das Objektsystem Τελος basiert auf Klassen und generischen Funktionen mit einfachem Standardverhalten und es stellt ein flexibles Metaobjektprotokoll zur Erweiterung dieses Verhaltens zur Verfügung.

Alle Sprachelemente bis auf Module, Bindungen und syntaktische Konstrukte treten als Objekte erster Ordnung in Erscheinung. Eine Bindung kann zur Laufzeit durch Berechnung eines Identifikators weder erzeugt noch modifiziert werden. Funktionen können zwar dynamisch zur Laufzeit generiert werden, ihr Quellcode muß aber zur Übersetzungszeit bekannt sein. EuLisp ist im Gegensatz zu CommonLisp auf Modul- und Applikations-Kompilation ausgerichtet sowie auf eine möglichst effiziente Programmausführung. Natürlich kann auch inkrementell übersetzt werden, um die Entwicklung von Software in lisp-typischer Weise zu unterstützen.

Das Objektsystem Τελος stellt einen integralen Bestandteil der EuLisp-Sprachdefinition dar und kann nicht von den restlichen Sprachbestandteilen isoliert betrachtet werden. Die verschiedenen Sprachkonzepte von EuLisp sind auf einander bezogen und unterstützen das Software-Engineering durch ihr geschicktes Zusammenspiel. Dies kann mit einer Referenzimplementierung von Τελος in CommonLisp leider nur mit gewissen Abstrichen gezeigt werden.

8.2 Entwurf und Implementierung von TELOS 1.0

Im Entwurf von Τελος sollen möglichst viele Ziele und Designkriterien aus Kapitel 2 sowie möglichst alle Konzepte objektorienterter Programmierung aus Kapitel 4 berücksichtigt werden. Dies wird jedoch nicht auf einen Schlag mit einem monolytischen System erreicht, sondern durch einen modularen und erweiterbaren Systemansatz. Danach richtet sich auch die Darstellung in diesem Kapitel. Implementierungsfragen diskutiere ich zum einen im Hinblick auf meine Referenzimplementierung in CommonLisp, zum anderen aber auch im Hinblick auf Optimierungsmöglichkeiten der Modul- und Applikations-Kompilation.

Das Hauptziel meiner Implementierung von Τελος ist die Validierung der entwickelten Konzepte. Sie erfolgt aus praktischen Gründen in einem Common-Lisp-Entwicklungssystem und heißt daher Celos. Das Programm ist aber bis auf wenige Ausnahmen auch ein korrektes EuLisp-Programm. Diese Vorgehensweise sollte im Kontext des APPLY-Projekts [Bretthauer *et al.*, 1994] dafür sorgen, daß Τελος mit den beiden dort entwickelten Compilern CLiCC und Eu2C komplettkompilierbar wird. Im Prinzip ist dies auch gelungen. Leider war Eu2C für die praktische Verwendung noch zu instabil. Mit CLiCC konnte nicht nur Celos, sondern auch eine erweiternde Anwendung von Τελος, nämlich der objektorientierte Wissensrepräsentationsformalismus TINA [Kopp, 1996a], erfolgreich komplettkompiliert werden. Entgegen dem weitverbreiteten Vorurteil [] wurde damit erstmals nachgewiesen, daß ein Metaobjektprotokoll keinen Interpretierer oder

inkrementellen Compiler im Laufzeitsystem einer objektorientierten Programmiersprache benötigt. Unter den gesetzten Randbedingungen einer weitgehenden CommonLisp- und EuLisp-Konformität wurden in Celos weniger technische Programmoptimierungen als in MCS vorgenommen. Um so mehr bestätigen die Performanzmessungen, daß durch ein konsequentes Sprachdesign eine hohe Effizienz mit einfachen Mitteln erreicht werden kann. Mit Celos wurde gleichzeitig die Laufzeit- und die Speichereffizienz gegenüber MCS weiter verbessert. Insgesamt ist Celos mit seiner Performanz selbst den heutigen kommerziellen CLOS-Systemen überlegen (siehe Abschnitt 8.2.6).

8.2.1 Ziele und Vorgehensweise

Um mich nicht zu wiederholen, nenne ich schlagwortartig die wichtigsten Designziele und -Kriterien, die in Kapitel 2 ausführlich diskutiert wurden:

- Abstraktion
- Klassifikation
- Spezialisierung
- Reflektion
- Erweiterbarkeit
- Komposition
- Modularisierung
- Orthogonalität
- Einfachheit
- Robustheit
- Effizienz
- Verursacherprinzip

Erreicht werden sollen diese Ziele durch einen Lösungsansatz, der mit folgenden Faustregeln beschrieben werden kann:

- Einfache Sprachkonstrukte sind besser als komplizierte.
- Orthogonale Sprachkonstrukte sind überlappenden vorzuziehen.
- Anforderungen der Programm-Entwicklung und der Programm-Ausführung müssen unterschieden werden.
- Zeitintensive Berechnungen sollen möglichst zur Übersetzungszeit und sonst eher zur Ladezeit als zur Laufzeit erfolgen.
- Spracherweiterungen und ihre Verwendung müssen auch syntaktisch separierbar sein.

- Modulabhängigkeiten bzgl. der Übersetzung und bzgl. der Ausführung müssen differenziert werden.

Die folgenden Unterabschnitte sollen klarmachen, welche Konzepte aus Kapitel 4 in ΤΕΛΟΣ realisiert werden und welche nicht.

8.2.2 Modularisierung von TELOS

Die Modularisierung von ΤΕΛΟΣ erfolgt unter mehreren Gesichtspunkten. Alle Sprachkonstrukte von EuLisp werden grob in zwei Schichten aufgeteilt: `level-0` und `level-1`. Bezogen auf ΤΕΛΟΣ reflektieren sie die Objekt- und die Metaobjektebene, siehe Abbildung 8.1.

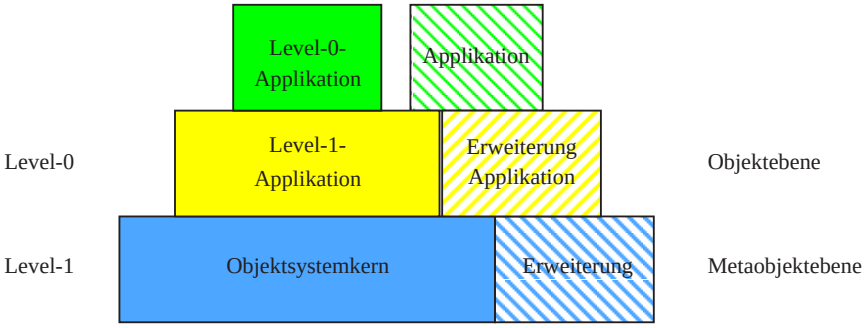


Abbildung 8.1: Schichten-Architektur von ΤΕΛΟΣ.

Die Benennung der Schichten reflektiert das Problem, eine metareflektive Sprache zu modularisieren. Bei der klassischen Schichtung von Sprachen befinden sich die primitiveren Schichten unten, die ausdrucksmächtigeren oben. Da die Sprachkonstrukte der Metaobjektebene ausdrucksstärker sind als die der Objektebene, ist es hier umgekehrt. Die Metaobjektebene stellt die Implementierungsmittel für die Objektebene zur Verfügung. Letztere kann als eine portable *Level-1-Applikation* implementiert werden, aber auch auf eine andere Art. Eine *Level-0-Applikation* kann dann von einem Compiler so übersetzt werden, daß zur Ausführungszeit keine Metaobjekte sowie ihre Operationen verwendet werden. Weiterhin kann es Applikationen geben, die die Metaobjektebene nur nutzen, z. B. für Introspektionsaufgaben, und solche, die sie erweitern und spezialisieren.

Aus Benutzersicht bietet die Objektebene von ΤΕΛΟΣ:

- systemdefinierte Klassen:
  - `<object>`,
  - `<list>`,
  - `<vector>`,
  - ...
- systemdefinierte Operationen:

  - `make`,
  - `allocate`,
  - `initialize`,
  - ...

- benutzerdefinierte Klassen mit
  - einfacher Vererbung,
  - spezialisierbaren Feldern und
  - spezialisierbaren `initialize`-Methoden;
- benutzerdefinierte generische Funktionen mit spezialisierbaren Methoden.

`Level-0` stellt aus Implementierersicht eine einfachere statisch analysierbare objektorientierte Sprache dar, die insbesondere von Applikations-Compilern effizient übersetzt werden kann. Globale Optimierungen wie Typinferenz und Inline-Kompilation können die Laufzeit von Applikationen mindestens um den Faktor zwei verbessern. Ob ein TELOS-Programm eine `level-0`-Applikation ist, bleibt nicht dem Zufall überlassen, sondern kann gezielt durch entsprechende Import-Ausdrücke garantiert werden. Wer Konstrukte von `level-1` nutzen will, muß sie auch importieren und sich der möglichen Konsequenzen für die Performanz bewußt sein. Die Benutzung von `level-1` soll damit nicht erschwert werden, es bedarf nur einer expliziten Entscheidung des Systemdesigners.

Die Metaobjektebene von TELOS bietet:

- systemdefinierte spezialisierbare Metaobjekt-Klassen:
  - `<class>`
  - `<slot>`
  - `<generic-function>`
  - `<method>`
  - ...
- systemdefiniertes Metaobjekt-Protokoll bzgl.:
  - Introspektion
  - Vererbung
  - Objekterzeugung
  - Feldzugriff
  - Methodenauswahl und generischem Dispatch
- benutzerdefinierte Metaobjekt-Klassen mit spezialisiertem Protokoll

`Level-1` zeichnet sich aus Implementierersicht dadurch aus, daß das spezifizierte Verhalten systemdefinierter Metaobjekte vom Benutzer nicht modifiziert werden kann. Dies ist nur für Spezialisierungen der Metaobjektklassen in vorgegebenem Rahmen möglich.

Darauf können sichere Optimierungen und ein garantiertes robustes Verhalten von Applikationen begründet werden. Die Modularisierung der Metaobjektebene erfolgt entlang den Teilprotokollen aus Kapitel 4. Sie unterstützt feinere Analysen der Abschlußeigenschaften von Programmen und somit noch besser angepaßte Optimierungen, je nachdem welches Teilprotokoll vom Benutzer verwendet bzw. spezialisiert wird. Abbildung 8.2 zeigt die modulare Architektur der Metaobjektebene von TELOS. Dabei wird deutlich, daß die Modularisierung der Teilprotokolle orthogonal zur Klassifikation der Metaobjekte erfolgt.

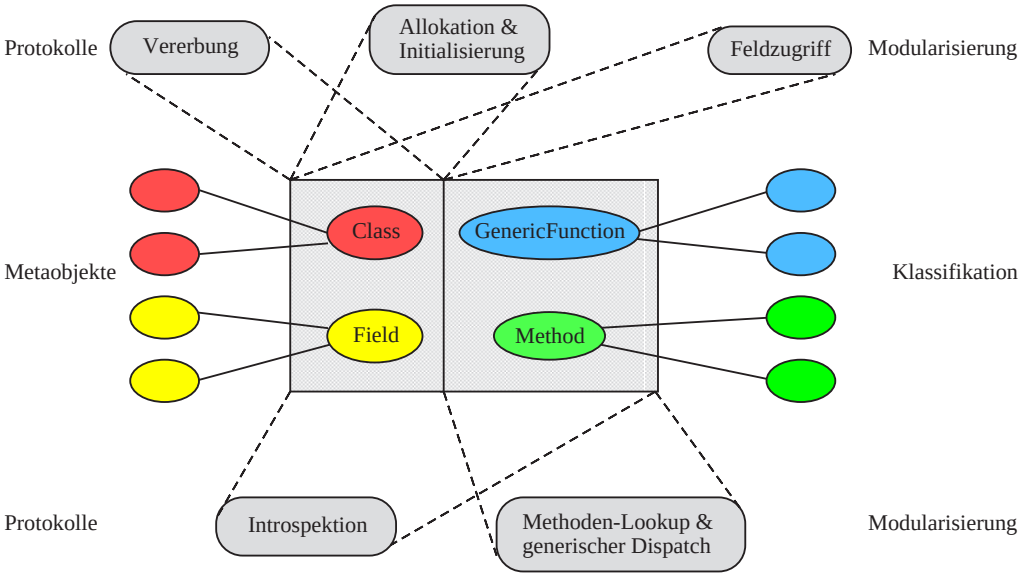


Abbildung 8.2: Modularisierung der Metaobjektebene von TELOS.

8.2.3 Objektebene

Die Beschreibung der Objektebene von TELOS beginne ich mit dem Überblick der Klassenhierarchie. Im nächsten Schritt werden die syntaktischen Sprachkonstrukte zur Definition von Klassen, generischen Funktionen und Methoden erläutert. Anschließend beschreibe ich die Operationen der Objektebene.

Klassenhierarchie der Objektebene

Die Objektebene von TELOS unterstützt zunächst nur einfache Vererbung, wie sie in Kapitel 4.3.1 auf Seite 86 beschrieben wurde. Auch die systemdefinierten Klassen sind nach einfacher Vererbung organisiert. Mixin-Vererbung und allgemeine multiple Vererbung können als Spezialisierung der Metaobjektebene bereitgestellt werden.

Abbildung 8.3 zeigt die Klassenhierarchie der Objektebene. Dabei wird durch Einrückung nach rechts die Subklassenbeziehung ausgedrückt. Da TELOS im Kern nach einfacher Vererbung organisiert ist, reicht dieses Ausdrucksmittel aus.

In EuLisp wurden Klassen und Lisp-Basistypen vollständig integriert. Die meisten Klassen der Objektebene entsprechen den Basistypen aus anderen Lisp-Dialekten. Die Klasse

```
<object>a
   <character>c
   <collection>a
      <sequence>a
         <character-sequence>a
            <string>c
         <list>a
            <cons>c
            <null>c
         <vector>c
      <table>a
         <hash-table>c
   <condition>a
      <telos-condition>a
         <no-next-method>c
         <non-congruent-lambda-lists>c
         <incompatible-method-domain>c
         <no-appliable-method>c
         <method-domain-clash>c
      ...
   <function>a
      <continuation>c
      <simple-generic-function>c
      <simple-function>c
   <lock>a
   <name>a
      <symbol>c
      <keyword>c
   <number>a
      <float>a
         <double-float>c
      <integer>a
         <fixed-precision-integer>c
         <variable-precision-integer>c
   <stream>a
      <char-file-stream>c
      <string-stream>c
```

Abbildung 8.3: Klassenhierarchie der Objektebene.

`<object>` bildet die Wurzel der Vererbungshierarchie, sie entspricht der Klasse `Object` aus Kapitel 4. Generell finden sich in ΤΕΛΟΣ die meisten Klassen und Operationen aus Kapitel 4 mit syntaktisch leicht veränderten Namen wieder. Klassennamen erhalten spitze Klammern, zusammengesetzte Namen werden mit Bindestrichen versehen und es wird alles klein geschrieben. Natürlich hätte ich in Kapitel 4 auch schon die Namenskonventionen von EuLisp verwenden können. Da ich jedoch davon ausgehe, daß die Java-Konventionen vielen Lesern geläufiger sind und da es eigentlich auf die Konzepte ankommt, muß der Notationswechsel an dieser Stelle in Kauf genommen werden.

Insgesamt werden alle Klassen in zwei Arten eingeteilt. Abstrakte Klassen, die keine direkte Instanzen besitzen tragen die Kennzeichnung ^{a}, während direkt instanziierbare Klassen, die auch als *konkrete* Klassen bezeichnet werden, mit ^{c} gekennzeichnet werden. In ΤΕΛΟΣ sind nur die abstrakten Klassen spezialisierbar. Konkrete Klassen besitzen keine Subklassen, sie sind somit abgeschlossen und können besser optimiert werden. Will man wiederverwendbare Software-Komponenten realisieren, so sollte man möglichst viele Methoden für abstrakte Klassen definieren. Konkrete Klassen sollten dann in der Regel nur der Instanzerzeugung dienen. Und ihre im Ausnahmefall definierten Methoden dienen primär weiteren Effizienzverbesserungen. Diese Designentscheidung in ΤΕΛΟΣ sollte aber nicht dogmatisch gesehen werden. Ob eine Klasse spezialisierbar ist, könnte man auch als eine weitere Klassenoption behandeln, d. h. von der Instanziierbarkeit abkoppeln. Die getroffene Entscheidung trägt zur Vereinfachung bei.

Die Klassenhierarchie deutet an, wie das Objektsystem und das *Condition-System*, d. h. die Ausnahmebehandlung, zusammenhängen. Für die jeweilige Ausnahmesituation ist eine Condition-Klasse definiert. Soll eine besondere Situation gemeldet werden, so wird `signal` mit einer neuen Instanz der entsprechenden Condition-Klasse sowie weiteren Argumenten aufgerufen. Um Ausnahmesituationen abzufangen und zu behandeln, gibt es das Konstrukt with-handler. Dabei spezifiziert man möglicherweise eine generische *Handler-Funktion* und die zu schützenden Ausdrücke. Natürlich gibt es auch ΤΕΛΟΣ-bezogene Ausnahmesituationen, für die eine gemeinsame Superklasse `<telos-condition>` als Subklasse von `<condition>` definiert wurde. Hier zeigt sich, wie die orthogonalen Sprachkonzepte des Objekt- und des Condition-Systems auf einander abgestimmt sind und wie sie von einander profitieren. In dieser Arbeit werde ich jedoch nicht weiter darauf eingehen.

Definieren von Klassen

Auf der Objektebene können Klassen nur statisch definiert werden. Das syntaktische Konstrukt `defclass` muß sich textuell unbedingt auf *Top-Level* eines Moduls befinden. Alle benutzerdefinierten Klassen stehen somit zur Übersetzungszeit fest und sind einem Applikations-Compiler auch alle bekannt. Von den in Kapitel 4 diskutierten Klassen- und Feldannotationen wurden einige aus Vereinfachungsgründen nicht realisiert. Dazu gehören die Feldannotationen `type:`, `initFunction:` und `initValue:`. Die Annotation `initExpression:` wird hier `default` genannt. Die genaue Syntax von `defclass` zeigt Abbildung 8.4.

Ursprünglich gab es in ΤΕΛΟΣ zwei Konstrukte, um Klassen zu definieren. Auf der Objektebene gab es `defstruct`, das sich von `defclass` der Metaobjektebene dadurch unterschied,

[2] Die Zeichen `|`, `{`, `}`, `[`, `]`, $^+$ und * haben die aus der Backus-Naur-Form bekannte Bedeutung.

```
(defclass class-name (superclass-name+)
  (slot-description*)
  class-option* )

slot-description ::= identifier | ([ slot-qualifier ] identifier slot-option*)

slot-qualifier ::= define: | specialize:
slot-option ::= keyword: identifier |
                required-keyword: identifier |
                default: lisp-form |
                reader: reader-name |
                writer: writer-name |
                accessor: reader-name |

class-option ::= keywords: (identifier+) |
                 initialize: method-description |
                 constructor: (constructor-name identifier*) |
                 predicate: predicate-name |
method-description ::= (specialized-lambda-list method-form*)

specialized-lambda-list ::= (specialized-parameter+ [ . identifier] )

specialized-parameter ::= (identifier class-name)

method-form ::= (next-method-p) | (call-next-method) | lisp-form
```

Abbildung 8.4: Syntax der Klassendefinition auf der Objektebene.[2]

daß es die Klassenoption `class:` für `defstruct` nicht gab. Um während der Programmentwicklung ohne umfangreiche Änderungen im Quellcode zwischen beiden wechseln zu können, werden jetzt beide mit `defclass` bezeichnet. Welches Konstrukt man tatsächlich benutzt, hängt davon ab, ob man im gegebenen Modul Konstrukte der Objektebene oder der Metaobjektebene importiert. Der *Rename*-Mechanismus von Modulen, erlaubt aber auch eine einfache Umbenennung von `defclass` der Objektebene z. B. in `defstruct`.

Eine einfache Lösung, `defclass` zu implementieren, ist die eines Makros. Ein angewandtes Vorkommen von `defclass` expandiert dann wie folgt:

```
(defclass <point> (<graphical-object>)
  ((define: x default: 0 accessor: point-x keyword: x:)
   (specialize: y default: 0 accessor: point-y keyword: y:))
  abstract: t
  initialize: (((obj <point>)  initlist) ...))

=>

(progn
  (defslotname x <point>)
  (defkeyword x: <point>)
  (defconstant <point>
               (make <simple-class>
                     name: '<point>
                     direct-superclasses: (list <graphical-object>)
                     defined-slot-descriptions:
                     (list (list keyword: x:
                                 default-function: (lambda () 0)
                                 name: x))
                     specialized-slot-descriptions:
                     (list (list keyword: y:
                                 default-function: (lambda () 0)
                                 name: y))
                     direct-keywords: (list x: y:)
                     abstractp: 't))
  (definitializer <point> ((obj <point>)  initlist) ...)
  (defreader point-x <point> x)
  ...
  <point>)
```

Nach dem ersten Expansionsschritt kommen im Ergebnis weitere definierende Konstrukte vor, die anschließend expandiert werden, wenn sie auch als Makros implementiert sind. Hier sind es die Implementierungshilfsmittel `defslotname`, `defkeyword`, `defreader`, `definitializer` etc., während `defconstant` ein Primitiv der EuLisp-Basissprache ist. Es erzeugt eine konstante Bindung im Modul, das die Definition textuell enthält. Hier können alle definierenden Hilfskonstrukte bis auf `definitializer` in `defconstant`-Ausdrücke expandieren. Für `defreader` kann dies beispielsweise wie folgt aussehen:

```
(defreader point-x <point> x) =>

(defconstant point-x (slot-reader (find-slot <point> x)))
```

Dies liegt daran, daß EuLisp *einen* Namensraum für Funktionen und alle anderen Objekte besitzt. Es handelt sich wie bei Scheme um ein sogenanntes *Lisp-1*, im Unterschied zu CommonLisp, das verschiedene Namensräume verwendet, und daher als *Lisp-2* bezeichnet wird, obwohl es in Wirklichkeit mindestens ein *Lisp-3* ist, da Klassen auch einen eigenen Namensraum bilden. Aus Sprachdesignsicht spricht nichts gegen mehrere Bindungsräume. Im Gegenteil, auf diese Weise kann für nicht-streng-typisierte Sprachen ein deutlich höheres Maß an Typsicherheit erreicht werden. Voraussetzung ist allerdings, daß diese Bindungsräume auch konstante Bindungen unterstützen und keine unzulässigen Manipulationen der Bindungen erlauben. Referenzen mit zur Laufzeit berechneten Bezeichnern sollten nicht zugelassen sein. Nur so kann ein Compiler auf explizite Bindungen im Laufzeitsystem sicher verzichten. Dadurch können erhebliche Effizienzsteigerungen erreicht werden. Τελος wird so entworfen, daß dies möglich ist.

Mit der Einführung der Klassenoption `initialize:` gibt es eigentlich zwei Varianten, `defclass` als Makro zu implementieren. In der ersten Variante wird die `initialize:`-Option, wie oben gezeigt, außerhalb der Initialisierung des Klassenobjekts behandelt. Zum Zeitpunkt der Erzeugung der Initialisierungsmethode ist dann die Klasse schon vorhanden, um als Referenz in der Domain-Spezifikation der Initialisierungsmethode referiert zu werden. Will man die `initialize:`-Option schon während der Klasseninitialisierung behandeln, muß das entsprechende Methodenobjekt erzeugt werden, bevor die Klasse initialisiert wird. Um das Methodenobjekt zu erzeugen, muß die Klasse schon existieren. Die Lösung dieses Henne-Ei-Problems besteht darin, das Allozieren des Klassenobjekts von seiner Initialisierung abzutrennen. Ein angewandtes Vorkommen von `defclass` expandiert dann wie folgt:

```
(defclass <point> (<graphical-object>)
  ((define: x default: 0 accessor: point-x keyword: x:)
   (specialize: y default: 0 accessor: point-y keyword: y:))
  abstract: t
  initialize: (((obj <point>)  initlist) ...))

=>

(progn
  (defslotname x <point>)
  (defkeyword x: <point>)
  (defconstant <point> (allocate <simple-class> '()))
  (initialize <point>
              (list name: '<point>
                    ...
                    initializer: (method-lambda ((obj <point>) initlist)
                                   ...)))
  (defreader point-x <point> x)
```

```
...
<point>)
```

Das Implementierungskonstrukt `definitializer` entfällt in dieser Variante. Stattdessen gibt es das neu Initialisierungsschlüsselwort `initializer:` gefolgt von einem entsprechenden Methodenobjekt, das in der Klasseninitialisierungsmethode behandelt werden kann.

Natürlich kann `defclass` auch in einen Aufruf eines speziellen Konstruktors expandieren, wenn es sich z. B. um Standardklassen, d. h. Instanzen von `<simple-class>`, handelt. Dann können spezielle Optimierungen zur Übersetzungszeit erfolgen. Aber auch die oben gewählte Variante läßt Compiler-Optimierungen zu. Eine sehr generische Optimierungstechnik ist die der Teilauswertung zur Übersetzungszeit (engl. *partial evaluation*). Dabei wertet der Compiler zur Übersetzungszeit solche Ausdrücke aus, die ein äquivalentes Ergebnis zur Lade- bzw. Laufzeit liefern würden. Dafür ist es nützlich, wenn möglichst viele Bezeichner konstant gebunden werden. Kann ihre Initialisierungsform zur Übersetzungszeit ausgewertet werden, so kann auch jedes angewandte Vorkommen des Bezeichners sicher durch das Ergebnis ersetzt werden. Auf diese Weise können weitere Teilausdrücke ausgewertet werden usw. Im obigen Beispiel kann `make`, angewandt auf `<simple-class>` zur Übersetzungszeit vollständig ausgeführt werden, da keine Abhängigkeiten von Variablen zur Laufzeit gegeben sind.

Definieren von generischen Funktionen und Methoden

In TELOS müssen generische Funktionen explizit definiert werden. Ein implizites Definieren über das erste Vorkommen von `defmethod` für einen gegebenen Bezeichner wie in CLOS wird nicht erlaubt. Dies ist wichtig, um z. B. ein von der Ladereihenfolge der Module unabhängiges Verhalten von Applikationen sicherzustellen. Wenn man eine Methode definiert, muß man sicher sein, ob es diese generische Funktion bereits gibt, ob die Parameterliste der neuen Methode mit der generischen Funktionsdefinition übereinstimmt und ob es eine Methode mit dem gleichen Bereich ggf. schon gibt oder nicht. Das Verhalten von `defclass` in Bezug auf Lese- und Schreiboperationen ist darauf abgestimmt. Jede Spezifikation einer der Slotoptionen `reader`, `writer` oder `accessor` führt zu einer neuen Bindung. Importierte Bindungen mit gleichen Namen führen zu einem Fehler, auch wenn es geerbte (und importierte) Lese- und Schreiboperationen sind.

Wie auch `defclass`, erzeugt `defgeneric` eine konstante Bindung des Bezeichners an ein generisches Funktionsobjekt, die vom Benutzer nicht verändert werden darf. Dies verbessert die Robustheit und erlaubt einige sichere Optimierungen, die auch nicht ungültig werden können, im Unterschied zu CLOS.

Für die Definition von Methoden gelten die Festlegungen aus Kapitel 4: `defmethod` erzeugt selbst keine Modulbindung, sondern meldet einen Fehler, falls es noch keine generische Funktion mit dem spezifizierten Bezeichner gibt. Das Ersetzen bereits definierter Methoden wird als Funktionalität der Programmentwicklungsumgebung betrachtet und in der Sprachspezifikation nicht unterstützt. Da bereits im Konstrukt `defgeneric` der Bereich einer generischen Funktion spezifiziert werden kann, gilt dies als Einschränkung für alle ihre Methoden, deren Argumentklassen spezieller oder gleich denen der generischen Funktion sein müssen. Das Konstrukt `defmethod` muß also dafür sorgen, daß entsprechende

`(defgeneric` *generic-function-name specialized-lambda-list*
generic-function-option* `)`

specialized-lambda-list `::= (`*specialized-parameter*$^+$ `[ .` *identifier*`] )`

specialized-parameter `::= (`*identifier class-name*`)`

generic-function-option `::= method: (`*method-description*`)`

method-description `::= (`*specialized-lambda-list method-form**`)`

method-form `::= (next-method-p) | (call-next-method) |` *lisp-form*

Abbildung 8.5: Syntax der Definition generischer Funktionen auf der Objektebene.[2]

Prüfungen gemacht werden. Dies verbessert die Robustheit von Anwendungen erheblich. Da man weiterhin sicher sein kann, daß definierte Methoden einer generischen Funktion nicht entfernt werden, können auch semantisch fundierte Optimierungen vorgenommen werden. Wie schon bei MCS 1.0 akzeptiert das spezielle Konstrukt `call-next-method` keine Argumente. Mit `next-method-p` kann festgestellt werden, ob es eine nächstallgemeinere Methode gibt. Im Abschnitt 8.3.4 diskutiere ich einige Erweiterungen dieses einfachen Konzepts der Methodenkombination.

`(defmethod` *generic-function-name specialized-lambda-list method-form** `)`

specialized-lambda-list `::= (`*specialized-parameter*$^+$ `[ .` *identifier*`] )`

specialized-parameter `::= (`*identifier class-name*`)`

method-form `::= (next-method-p) | (call-next-method) |` *lisp-form*

Abbildung 8.6: Syntax der Methodendefinition auf der Objektebene.[2]

Aus Benutzersicht sollten sich generische Funktionen so verhalten, wie es die regulären (nicht-generischen) Funktionen auch tun. Wie in CommonLisp gibt es in EuLisp nicht nur über das Konstrukt `defun` modul-definierte Funktionen, sondern auch anonyme lokale Funktionen, die mit `lambda`[3] erzeugt werden. Semantisch kann `defun` auf die Konstrukte `defconstant` und `lambda` zurückgeführt werden:

```
(defun f (...) ...)  =>  (defconstant f (lambda (...) ...))
```

Für generische Funktionen gibt es daher ein entsprechendes Konstrukt, um lokale generische Funktionen zu erzeugen. Somit läßt sich `defgeneric` auf `generic-lambda` und `defconstant` zurückführen:

```
(defgeneric gf (...) ...)  =>  (defconstant gf (generic-lambda (...) ...))
```

Die Syntax von `generic-lambda` entspricht, bis auf die Benennung, der Syntax von `defgeneric`:

`(generic-lambda` *specialized-lambda-list generic-function-option** `)`

Auf der Metaobjektebene wird `generic-lambda` nicht nur aus Uniformitätsgründen benötigt, sondern auch, um lokale anonyme generische Lese- und Schreiboperationen zu generieren (siehe Unterabschnitt 8.2.4. Die semantische Rückführung von Sprachkonstrukten auf primitivere (engl. *rewrite rules*) ist für eine Implementierung von EuLisp nicht zwingend. Sie erlaubt aber eine einfache und klare Semantikspezifikation der Konstrukte.

[3] In CommonLisp mit `(function (lambda (...) ...))`.

(`make` *cl initarg**) → *object*
(`allocate` *cl initargs*) → *object*
(`initialize` *object initargs*) → *object*

Abbildung 8.7: Allokations- und Initialisierungsoperationen der Objektebene.

Methodenauswahl und generischer Dispatch

Auf der Objektebene unterstützt ΤΕΛΟΣ einen klassenbasierten generischen Dispatch auch über mehrere Argumente. `eql`-Methoden wie in CLOS werden nicht unterstützt. Sie können aber auf der Metaobjektebene als Erweiterung von ΤΕΛΟΣ sogar portabel implementiert werden. Das Konzept der Methodenauswahl und der generische Dispatch entsprechen genau dem Verfahren aus Kapitel 4.2.2.

Allozieren und Initialisieren von Objekten

Das Allozieren und Initialisieren von Objekten in der veröffentlichten Version von ΤΕΛΟΣ [Padget *et al.*, 1993] entspricht dem Konzept in Kapitel 4.2.3 auf Seite 83. Danach gibt es die zwei generischen Funktionen `allocate` und `initialize`, wobei nur `initialize` vom Benutzer spezialisiert werden kann. Die nicht-generische Funktion `make` ruft `allocate` und `initialize` auf. Daß `allocate` auf der Objektebene nicht spezialisiert werden kann, wird auf einfache Weise sichergestellt: Die Klasse des ersten Parameters muß eine Subklasse von `<class>` sein. Da es auf der Objektebene keine Möglichkeit gibt, Subklassen von `<class>` zu definieren, können auch keine `allocate`-Methoden hinzukommen.

Hier habe ich mich für eine neue Lösung entschieden, die als zweite Alternative in Kapitel 4.4.6 auf Seite 108 bereits diskutiert wurde. Auf der Objektebene von ΤΕΛΟΣ muß sich nicht viel ändern. Aus Benutzersicht kann man nach wie vor die Operationen `make`, `allocate` und `initialize` wie gewohnt aufrufen. Lediglich das Spezifizieren von Initialisierungs-Methoden muß textuell innerhalb von `defclass` erfolgen. Dies wurde in der Syntaxspezifikation von `defclass` in Abbildung 8.4 mit der Klassenoption `initialize:` schon berücksichtigt. Die `initialize`-Spezifikation gleicht syntaktisch einer Methodenspezifikation innerhalb von `defgeneric`. Natürlich kann man hier auch eine andere syntaktische Form festlegen. Wichtig ist, daß man im Rumpf einer Initialisierungs-Methode die nächstallgemeinere Methode mit `call-next-method` wie in regulären Methoden aufrufen kann. Dies muß die aus Benutzersicht nicht-generische Funktion `initialize` ähnlich den generischen Funktionen organisieren, siehe Abschnitt 8.2.4 auf Seite 243. Abbildung 8.7 zeigt die Operationen der Objektebene im Überblick.

Aus Implementierungssicht ist entscheidend, daß Klassenobjekte die Abschlußeigenschaft bzgl. der Instanzerzeugung aufweisen. Dadurch kann das Allozieren und Initialisieren von Objekten in jeder Implementierungsart transparent und sicher optimiert werden: von der portablen Einbettung über inkrementelle und Modul- bis hin zur Applikations-Kompilation. Einer Implementierung ist es nicht verwährt, `allocate` und `initialize` als generische Funktionen zu realisieren und die Klassenannotation `initialize` auf `defmethod` zurückzuführen. Deshalb wurde oben im Expansionsergebnis von `defclass` die Form

`(definitializer ...)` aufgeführt. Der Unterschied zwischen Initialisierungs-Methoden und regulären Methoden generischer Funktionen besteht darin, daß Initialisierungs-Methoden textuell nicht von `defclass` getrennt definiert werden sollen.

Im Kontext der Modul-Kompilation wäre es auch denkbar, die Moduldefinition als syntaktische Abschluß-Klammer festzulegen, so daß Initialisierungs-Methoden zwar über `defmethod` definiert werden können, dies aber textuell im selben Modul wie die Klassendefinition erfolgen muß. Da ich hier die Einbettungstechnik verwende, habe ich die Modullösung jedoch verworfen.

Feldzugriffe

Auf der Objektebene unterstützt ΤΕΛΟΣ automatisch generierte Lese- und Schreiboperationen. Bei der Klassendefinition werden sie an spezifizierte Bezeichner gebunden. Da auf Objektebene nur einfache Vererbung zur Verfügung steht, werden die Lese- und Schreiboperationen als nicht-generische Funktionen realisiert. Dies entspricht meinen Ausführungen in Kapitel 4.2.4 auf Seite 85.

8.2.4 Metaobjektebene

Die Metaobjektebene von ΤΕΛΟΣ kann man als die veröffentlichte Schnittstelle der Implementierung der Objektebene ansehen. Metaobjekte sind Implementierungsobjekte, die dafür sorgen, daß die Mechanismen der Objektebene funktionieren. Der Sprachumfang der Objektebene wird daduch erweitert, daß Metaobjektklassen spezialisiert werden. Daher ist die Objektebene von ΤΕΛΟΣ bewußt klein und einfach gehalten. Die Ausdruckskraft liegt im einfach spezialisierbarem Metaobjektprotokoll, das ein breites Spektrum von objektorientierten Konzepten abdeckt. Die Funktionalität von ΤΕΛΟΣ kann durch portable ΤΕΛΟΣ-Programme robust und effizient erweitert werden.

Wie schon die Objektebene entspricht die Metaobjektebene im wesentlichen den Konzepten aus Kapitel 4. Hier beschreibe ich einige Abweichungen und vertiefe die Implementierungsaspekte des Metaobjektprotokolls.

Klassenhierarchie der Metaobjektebene

Abbildung 8.8 zeigt die Klassenhierarchie der Metaobjektebene. Wie schon in Abbildung 8.3 wird durch Einrückung nach rechts die Subklassenbeziehung ausgedrückt. Klassen der Objekteben wiederhole ich hier nur, wenn sie zur Implementierung auch benötigt werden, z. B. `<object>`, `<vector>`, `<list>` etc.

Die obere Hälfte der Klassenhierarchie zeigt die eigentlichen Metaobjektklassen, während die unteren Klassen verwendet werden, um das Metaobjektprotokoll zu implementieren. Die Klassenhierarchie zeigt auch einige Abweichungen zu Kapitel 4.4.1 auf Seite 97. Es gibt keine explizite Klasse `<metaobject>`, weil es keine Felder oder Methoden gibt, die man für sie definieren würde. Im Unterschied zur veröffentlichten Version von ΤΕΛΟΣ in [Padget *et al.*, 1993] sind Methoden auch funktionale Objekte, wie generische Funktionen. Daher ist `<method>` eine Subklasse von `<function>`. Methoden unterscheiden sich von einfachen Funktionen dadurch, daß ihnen ihr Bereich (engl. *domain*) zugeordnet ist.

```
<object>a
   <class>a
      <built-in-class>a
      <function-class>c
      <simple-class>c
   <slot>a
      <local-slot>c
   <function>a
      <simple-function>c
      <generic-function>a
         <simple-generic-function>c
      <method>a
         <simple-method>c
      <continuation>c

   <condition>a
      <telos-condition>a
         <no-next-method>c
         <non-congruent-lambda-lists>c
         <incompatible-method-domain>c
         <no-appliable-method>c
         <method-domain-clash>c
      ...
   <symbol>c
   <list>a
      <cons>c
      <null>c
   <vector>c
   <hash-table>c
   <fixed-precision-integer>c
```

Abbildung 8.8: Klassenhierarchie der Metaobjektebene.

```
(class-name class) → symbol
(class-precedence-list class) → list
(class-slots class) → list
(class-keywords class) → list
(class-abstractp class) → boolean
(class-instance-size class) → integer
(class-allocator class) → function
(class-initialize-methods class) → list

(slot-name slot) → symbol
(slot-default-function slot) → function
(slot-reader slot) → function
(slot-writer slot) → function
(slot-keyword slot) → symbol
(slot-required-keyword-p slot) → boolean
```

Abbildung 8.9: Introspektionsoperationen für Klassen und Felder.

```
(generic-function-name gf) → symbol
(generic-function-domain gf) → list
(generic-function-method-class gf) → class
(generic-function-methods gf) → list
(generic-function-discriminating-function gf) → function

(method-domain method) → list
```

Abbildung 8.10: Introspektionsoperationen für generische Funktionen und Methoden.

Introspektionsprotokoll

Abbildungen 8.9 und 8.10 zeigen die Introspektionsoperationen für Klassen, Felder, generische Funktionen und Methoden im Überblick.

Auch hier gelten im Übrigen die Festlegungen aus Kapitel 4.4.2 auf Seite 98. Alle Introspektionsoperationen sind in ΤΕΛΟΣ nicht-generisch. Es werden auch keine Felder der Metaobjektklassen spezifiziert. Dies beschränkt zwar die Möglichkeiten, die Repräsentation von Metaobjekten zu spezialisieren, erlaubt aber wesentlich effizientere und sicherere Zugriffe. Gerade für benutzerdefinierte Metaobjekte entfällt die aufwendige Unterbrechung potentieller Zyklen im dynamischen Ablauf des Introspektionsprotokolls im Vergleich zu CLOS [Kiczales *et al.*, 1991, S. 269 ff].

Erzeugen und Initialisieren von Metaobjekten

Auf der Metaobjektebene muß spezifiziert werden, wie man Metaobjekte erzeugen und initialisieren kann. Hier unterscheide ich zwei Arten von Metaobjekten:

```
name: symbol
direct-superclasses: list
defined-slot-descriptions: list
specialized-slot-descriptions: list
direct-keywords: list
abstractp: boolean
initialize-method: method
```

Abbildung 8.11: Initialisierungsschlüsselwörter der Klasse `<class>`.

```
name: symbol
default-function: function
reader: function
writer: function
keyword: keyword
required-keyword-p: boolean
```

Abbildung 8.12: Initialisierungsschlüsselwörter der Klasse `<slot>`.

- *Statische Objekte*, die aus anderen Objekten zusammengesetzt werden und die nicht neue ausführbare Funktionen repräsentieren, können über die funktionale Schnittstelle der Operationen `make`, `allocate` und `initialize` erzeugt und initialisiert werden. Hierzu zählen Klassen- und Feldobjekte.
- Die Erzeugung *funktionaler Objekte*, verbunden mit der Generierung von Funktionen mit neuem Codemuster, muß über eine syntaktische Schnittstelle erfolgen. Im letzteren Fall ist eine Aufspaltung in Allozieren und Initialisieren nicht immer möglich und sinnvoll. Dies betrifft generische Funktionen und Methoden, für die es die speziellen Konstruktoren `generic-lambda` und `method-lambda` gibt. Wollte man im Rumpf einer Lisp-Funktion ein neues funktionales Objekt erzeugen, dessen Codemuster in Form einer Liste als Argument durchgereicht wird, so müßte man explizit Programmtext in Code umwandeln. Gerade dies schließe ich in ΤΕΛΟΣ aus, um das Metaobjektprotokoll mit der Komplett-Kompilation verträglich zu gestalten. Dies schließt aber nicht aus, daß es benutzerspezifizierte Initialisierungsmethoden auch für selbst definierte funktionale Metaobjektklassen geben darf. Sie können nur nicht explizit ausgeführt werden.

Für beide Arten der Objekterzeugung werden gleichermaßen Initialisierungsschlüsselwörter spezifiziert, die in benutzerdefinierten Metaobjektklassen erweitert werden können. Die Abbildungen 8.11, 8.12, 8.13 zeigen, welche Initialisierungsschlüsselwörter für systemdefinierte Metaobjektklassen spezifiziert wurden. Für die Metaobjektklasse `<method>` gibt es nur das Initialisierungsschlüsselwort `domain:`.

Hierzu ist anzumerken, daß dies die Schlüsselwörter der Initialisierungsmethoden sind. Die syntaktischen Konstrukte `defclass`, `defgeneric`, `defmethod`, `generic-lambda` und

```
name: symbol
domain: list
method-class: class
methods: list
```

Abbildung 8.13: Initialisierungsschlüsselwörter der Klasse `<generic-function>`.

`method-lambda` besitzen ihre eigenen Schlüsselwörter. Die Syntax der ersten vier Konstrukte wurde für die Objektebene bereits auf den Seiten 229, 233 und 234 und 234 spezifiziert. Hier werden sie um einige Optionen erweitert, siehe Abbildungen 8.14, 8.15 und 8.16 im Abschnitt 8.2.4. Die Syntax von `method-lambda` entspricht bis auf den Namen der generischen Funktion, der die neue Methode hinzugefügt werden soll, der Syntax von `defmethod`:

`(method-lambda` *method-option*[*] *specialized-lambda-list* *method-form*[*] `)`

Syntaktische Erweiterungen der Metaobjektebene

Die syntaktischen Konstrukte von ΤΕΛΟΣ werden auf der Metaobjektebene um einige Optionen explizit erweitert. Darüberhinaus sind sie in der Lage, unbekannte Optionen an benutzerdefinierte Initialisierungsmethoden entsprechender Metaobjektklassen durchzureichen. Die zusätzlichen Optionen `class:` und `method-class:` in den Konstrukten `defclass`, `defgeneric`, `defmethod`, `generic-lambda` und `method-lambda` können nur auf der Metaobjektebene sinnvoll verwendet werden. Sie wurden in den syntaktischen Konstrukten auf der Objektebene daher nicht erwähnt.

Vererbungsprotokoll

Das Vererbungsprotokoll in ΤΕΛΟΣ entspricht voll dem Konzept in Kapitel 4.4.4 auf Seite 102. Nur die Benennungen der Operationen folgen den EuLisp-Konventionen. Aus Platzgründen verzichte ich hier darauf, diese syntaktische Transformation zu explizieren.

Feldzugriffsprotokoll

Für das Feldzugriffsprotokoll gilt das gleiche wie für das Vererbungsprotokoll. Es entspricht bis auf syntaktische Umbenennungen dem Konzept aus Kapitel 4.4.5 auf Seite 104.

Aus Implementierungssicht ist hier anzumerken, daß die vom Objektsystemkern definierten Methoden für die Berechnung von nicht-generischen Lese- und Schreiboperationen lediglich das Lisp-Basiskonstrukt `lambda` benötigen. Aber schon bei der portablen Erweiterung des Kerns um die Mixin-Vererbung, kann man ohne die Konstrukte `generic-lambda` und `method-lambda` nicht auskommen, jedenfalls nicht wenn man das vorgesehene Protokoll

[4] Die Zeichen `|`, `{`, `}`, `[`, `]`, `+` und `*` haben die aus der Backus-Naur-Form bekannte Bedeutung.

```
(defclass class-name (superclass-name+)
  (slot-description*)
  class-option* )

slot-description ::= identifier | ([ slot-qualifier ] identifier slot-option*)

slot-qualifier ::= define: | specialize:
slot-option ::= keyword: keyword |
                default: lisp-form |
                reader: reader-name |
                writer: writer-name |
                accessor: reader-name |
                keyword lisp-form

class-option ::= keywords: (keyword+) |
                 initialize: method-description |
                 constructor: (constructor-name keyword*) |
                 predicate: predicate-name |
                 class: metaclass-name |
                 keyword lisp-form

method-description ::= (specialized-lambda-list method-form*)

specialized-lambda-list ::= (specialized-parameter+ [ . identifier] )

specialized-parameter ::= (identifier class-name)

method-form ::= (next-method-p) | (call-next-method) | lisp-form
```

Abbildung 8.14: Syntax der Klassendefinition auf der Metaobjektebene.[4]

```
(defgeneric generic-function-name specialized-lambda-list
  generic-function-option* )

specialized-lambda-list ::= (specialized-parameter+ [ . identifier] )

specialized-parameter ::= (identifier class-name)

generic-function-option ::= class: generic-function-class-name
                            method-class: method-class-name |
                            method: (method-description) |
                            identifier lisp-form

method-description ::= (method-option*
                        specialized-lambda-list method-form*

method-option ::= class: method-class-name |
                  identifier lisp-form

method-form ::= (next-method-p) | (call-next-method) | lisp-form
```

Abbildung 8.15: Syntax der Definition generischer Funktionen auf der Metaobjektebene.[4]

```
(defmethod generic-function-name
  method-option*
  specialized-lambda-list
  method-form* )

method-option ::= class: method-class-name |
                  identifier lisp-form

specialized-lambda-list ::= (specialized-parameter+ [ . identifier] )

specialized-parameter ::= (identifier class-name)

method-form ::= (next-method-p) | (call-next-method) | lisp-form
```

Abbildung 8.16: Syntax der Methodendefinition auf der Metaobjektebene.[4]

`(make` *cl initarg**`)` → *object*
`(allocate` *cl initargs*`)` → *object*
`(initialize` *object initargs*`)` → *object*
`(compute-allocator` *cl initkeywords*`)` → *function*
`(compute-inherited-initialize-methods` *cl direct-superclasses*`)` → *list*
`(compute-initialize-methods` *cl direct-initialize-method inherited-initialize-methods*`)` → *list*

Abbildung 8.17: Allokations- und Initialisierungsoperationen der Metaobjektebene.

spezialisieren will. Ohne diese Konstrukte könnte es keine vollwertigen anonymen Klassenobjekte geben, eben weil man keine anonymen Lese- und Schreiboperationen berechnen kann.

Allokations- und Initialisierungsprotokoll

Um das neue Konzept der Objekterzeugung und -Initialisierung aus der Objektebene, siehe Seite 235, zu realisieren, werden auf der Metaobjektebene weitere Protokollfunktionen eingeführt, die in Kapitel 4.4.6 schon genannt wurden. Abbildung 8.17 zeigt die Operationen der Objekt- und Metaobjektebene im Überblick.

Die Grundidee ist die gleiche wie bei Feldzugriffsoperationen. Zur Klasseninitialisierungszeit werden aufwendigere Berechnungen durchgeführt, um klassenspezifisch optimierte Operationen bereitzustellen, die später zur Instanzerzeugungszeit effizient ausgeführt werden können.

Im Fall des *Allozierens* liefert `compute-allocator` als Ergebnis eine nicht-generische Funktion, die mit dem Klassenobjekt assoziiert wird. Später kann sie mit der nicht-generischen Introspektionsfunktion `class-allocator` geholt werden. Will man das Allokationsverhalten von Klassen spezialisieren, muß man eine neue Methode `compute-allocator` für die entsprechende Metaklasse, also auf Metaobjektebene, definieren.

Für das *Initialisieren* muß die Spezialisierbarkeit, inklusive des `call-next-method`-Mechanismus, auf der Objektebene gegeben sein. Auf der Objektebene werden daher Initialisierungs-Methoden spezifiziert. Auf der Metaobjektebene sorgt man dafür, daß die Initialisierungs-Methoden von den Superklassen ererbt und in einer geordneten Liste bereitgestellt werden. Im Standardfall kommt die speziellste Methode zuerst. Aber auch eine umgekehrte Reihenfolge kann in speziellen Metaklassen vorgegeben werde. Das Protokollmuster entspricht hier der Vererbung der Initialisierungsschlüsselwörter: Zur Klasseninitialisierungszeit werden mit `compute-inherited-initialize-methods` zuerst die Initialisierungs-Methoden der Superklassen berechnet bzw. geholt, was bei einfacher Vererbung mit einem Introspektionszugriff mittels `class-initialize-methods` erledigt ist. `compute-initialize-methods` braucht dann nur die ggf. direkt spezifizierte Initialisierungs-Methoden mit `cons` anfügen.

Die Implementierung dieses Konzepts ist also denkbar einfach:

```
(defmethod compute-inherited-initialize-methods ((cl <class>) direct-
superclasses)
```

```
  ;; assumptions: single inheritance and
  ;;             at least one direct superclass
  (class-initialize-methods (first direct-superclasses)))

(defmethod compute-initialize-methods ((cl <class>)
                                       direct-initialize-method
                                       inherited-initialize-methods)
  ;; assumptions: single inheritance and
  ;;             at least one inherited-initialize-
method, e.g. from <object>
  (if direct-initialize-method
    (cons direct-initialize-method inherited-initialize-methods)
    inherited-initialize-methods))
```

Für den Fall der multiplen oder der Mixin-Vererbung müssen die Initialisierungs-Methoden gemäß der Klassenpräzedenzliste eingesammelt werden.

Die Funktionen `make`, `allocate` und `initialize` können wie folgt implementiert werden:

```
(defun allocate (cl init-list)
  (funcall (class-allocator cl) cl init-list))

(defun initialize (object init-list)
  (let ((init-methods (class-initialize-methods (class-of object))))
    (funcall (first init-methods)    ; methods are funcallable
             init-methods            ; needed by call-next-method
             object
             init-list)

(defun make (cl . init-list)
  (let ((init-methods (class-initialize-methods cl)))
    (funcall (first init-methods)    ; methods are funcallable
             init-methods            ; needed by call-next-method
             (funcall (class-allocator cl) cl init-list)  ; the new object
             init-list)))
```

Das neue Protokoll liefert schon mit der einfachsten Implementierung eine effizientere Lösung als das ursprüngliche Konzept aus CLOS, aber auch als die vereinfachte Variante aus MCS 1.0 bzw. aus ΤΕΛΟΣ 0.99. Im Vergleich zu MCS 1.0 wird der zweifache generische Dispatch von `allocate` und `initialize` eingespart. Wesentlich weitergehende Optimierungen können transparent realisiert werden. Zum Beispiel kann der Allokator die Kopie einer prototypischen Instanz mit vorbelegten Initialisierungswerten zurückliefern. Die systemdefinierte Initialisierungs-Methode von `<object>` muß dann nur noch die erneut auszuwertenden Default-Ausdrücke berechnen und die expliziten Initialisierungsargumente behandeln.

Im Vergleich zu den optimierten Konstruktorfunktionen in MCS 0.5 hat die ΤΕΛΟΣ-Lösung den Vorteil, daß sie ein ebenso differenziertes Protokoll bietet wie MCS 1.0 bzw.

CLOS. Aus MCS 0.5 kann hier die Idee übernommen werden, für jede Klasse ein spezielles Codemuster zu berechnen, das semantisch äquivalent zum spezifizierten Standardverhalten ist, in dem aber möglichst viele Ergebnisse der zur Übersetzungszeit partiell ausgewerteten Ausdrücke eingesetzt sind. In MCS 0.5 wurde dafür zur Ladezeit der inkrementelle Compiler von CommonLisp aufgerufen, siehe Seite 171 und Seite 176. Dies ist in EuLisp auf portable Weise zurecht nicht möglich. Aber ein Modul- oder Applikations-Compiler für EuLisp kann diese Optimierungen natürlich durchführen.

Methodenauswahl und generischer Dispatch

Auf der Objektebene haben sich die Methodenauswahl und der generische Dispatch gegenüber dem Konzept aus Kapitel 4 nicht geändert. Die Lösung auf der Metaobjektebene läßt aber einen gewissen Entscheidungsspielraum offen.

Auf Seite 110 dieser Arbeit und schon in der veröffentlichten Version von Τελος [Padget *et al.*, 1993, S. 89] habe ich eine Variante beschrieben, die sich stärker an der Grundidee des Feldzugriffsprotokolls orientiert. Sie unterscheidet sich von der CLOS-Lösung in einem wichtigen Punkt. Für die Methodenauswahl wird mit der generischen Protokolloperation `compute-method-lookup-function` eine spezielle, nicht-generische *Lookup-Funktion* berechnet, die bei Ausführung einer generischen Funktion aufgerufen wird, falls kein Eintrag im Cache[5] gefunden wird. Somit muß zur Dispatchzeit im Cache-Fail-Fall nicht ein zweiter Dispatch durchgeführt werden, um den Cache-Eintrag zu berechnen.

Letzteres passiert aber in CLOS, wo es an Stelle von `compute-method-lookup-function` die generische Protokolloperation `compute-applicable-methods` gibt. Hier liegt der Vorteil darin, daß nicht für jede generische Funktion eine eigene Lookup-Funktion berechnet werden muß und somit Initialisierungszeit und Speicherplatz eingespart werden.

Unter der Voraussetzung, daß Closures als Lisp-Basisprimitiv effizient realisiert sind, ist die Τελος-Lösung vorzuziehen. Ihr weiterer Vorteil ist die Vermeidung einer potentiellen Endlosschleife im Dispatch von `compute-applicable-methods`, die in CLOS explizit aufgebrochen werden muß, siehe [Kiczales *et al.*, 1991, S. 269 ff]. In Τελος muß nur dafür gesorgt werden, daß `compute-method-lookup-function` manuell initialisiert wird, ohne daß eine generische Funktion ausgeführt wird.

Abschließend läßt sich festhalten, daß der generische Dispatch mit portablen Lisp-Basisprimitiven nur bedingt optimiert werden kann. Um mit Sprachen wie C++ mithalten zu können, müssen in einer kommerziellen Sprachimplementierung auch Optimierungen auf Maschinenebene erfolgen. Darauf gehe ich in dieser Arbeit nicht ein. Ich stelle jedoch sicher, daß vom Sprachdesign her alle bekannten Techniken wie statische Analyse, Typinferenz, Inlining, partielle Auswertung etc. erfolgreich angewandt werden können.

8.2.5 Unterschiede zwischen TELOS und CELOS

In der hier beschriebenen Version von Τελος gibt es gegenüber den postulierten Konzepten in Kapitel 4 kaum nennenswerte Abweichungen. Es gibt in Τελος beispielsweise

[5] Bei inkrementeller oder bei der Modul-Kompilation wird ein effizienter Dispatch immer auf die Cache-Technik hinauslaufen. Und auch bei der Applikations-Kompilation wird man nicht immer ohne Cache auskommen.

keine Klasse `<metaobject>`. Dies hat jedoch keine Auswirkung auf das Metaobjektprotokoll, weil `MetaObject` in Kapitel 4 eine rein didaktische Rolle spielt. Die Initialisierungsschlüsselwörter wurden hier entsprechend den EuLisp-Konventionen benannt.

Es sind aber einige Abweichungen meiner Referenzimplementierung Celos zu Τελος zu beachten:

- Bei der Definition von Klassen wird in `defclass` nicht explizit zwischen der Spezifikation neuer Felder und der Spezifikation speziellerer Annotationen für zu erbende Felder unterschieden. Die Syntaxspezifikation in Abbildung 8.4 unterscheidet beide Fälle über die Schlüsselwörter `define:` und `specialize:` vor dem Slotnamen. In meiner Referenzimplementierung habe ich diese Neuerung noch nicht umgesetzt. Dieser Vorschlag war in der EuLisp Working Group seinerzeit auch noch nicht konsensfähig.
- Da es in CommonLisp keine Module gibt, werden für Feldbezeichner und für Initialisierungsschlüsselwörter keine Modulbindungen generiert. Felder werden daher wie in CLOS über ihre spezifizierten Namenssymbole direkt identifiziert. Deshalb wird die Forderung nach Kapselung von Feldern und Initialisierungsschlüsselwörtern noch nicht hinreichend berücksichtigt. Man braucht EuLisp-Module, um das Konzept voll zu realisieren.

Trotz dieser Abweichungen wurden die objektorientierten Konzepte mit Τελος als Entwurf und mit Celos als Implementierung experimentell erfolgreich validiert. Dies bestätigen auch die Performanzmessungen, auf die ich im nächsten Abschnitt eingehe.

8.2.6 Performanzmessungen

Die folgenden Performanzmessungen beziehen sich auf meine Implementierung von Τελος in CommonLisp, genannt Celos. Um eine bessere visuelle Unterscheidung zu CLOS zu ermöglichen, spreche ich hier einfach von Τελος.

Die durchgeführten Laufzeit- und Speichertests orientieren sich an den Tests mit MCS 1.0. Auch hier bin ich von einem Verhältnis 1:10 von Klassen zu Methoden ausgegangen. Die Gewichtung der anderen Tests hat allerdings mit diesem Verhältnis nichts zu tun. Wie viele Instanzen pro Klasse erzeugt werden und wie oft generische Funktionen durchschnittlich aufgerufen werden, ist sehr anwendungsspezifisch. Die gezeigten Summen aller Tests in den Abbildungen 8.26, 8.27, 8.28 und 8.29 erheben daher nicht den Anspruch einer wie auch immer zu bestimmenden repräsentativen Gewichtung der Tests untereinander. Dennoch gibt die Summe einen Hinweis auf die Gesamtperformanz der Objektsysteme.

Beschreibung der Tests

Im Vergleich zu den Tests in Kapitel 7.3 wurde bei den folgenden Messungen dem Aspekt statischer Optimierungen zur Übersetzungszeit stärker Rechnung getragen: erstens stehen alle Klassen- und Methodendefinitionen textuell auf Top-Level des Testprogramms (**t1** und **t2**) und zweitens wurden zusätzlich die Tests **t3a** und **t4a** durchgeführt:

- **t1** erzeugt mit `defclass` auf Top-Level 10 Klassen.
- **t2** erzeugt auf Top-Level mit `defgeneric` 7 generische Funktionen und mit `defmethod` 10 Methoden.
- **t3** erzeugt mit einem dynamischen Aufruf von `make` bzw. `make-instance` zur Laufzeit 1000 terminale Instanzen einer Klasse. Hier sind die Argumente von `make` für den Compiler im allgemeinen nicht ableitbar.
- **t3a** erzeugt mit einem statischen Aufruf von `make` bzw. `make-instance` zur Laufzeit 1000 terminale Instanzen einer Klasse. Alle Argumente von `make` sind zur Übersetzungszeit bekannt und für den Compiler ableitbar. Die Aufrufe von Protokolloperationen können zur Ausführungszeit entfallen.
- **t4** greift mit `slot-value` 10000 mal dynamisch auf Slots zu. Hier sind die Argumente von `slot-value` für den Compiler im allgemeinen nicht ableitbar. Für ΤΕΛΟΣ ist hier anzumerken, daß es die generelle Operation `slot-value` im Kern aus konzeptionellen Gründen nicht bereitstellt. Für diesen Test wurde eine nicht-optimierte Funktion mit vorhandenen spezifizierten Mitteln implementiert, um dennoch einen Vergleich mit CLOS und MCS zu erhalten.
- **t4a** greift mit `slot-value` in einer Methode der Klasse, die den Slot definiert, 10000 mal auf einen Slot zu. Alle Argumente von `slot-value` sind zur Übersetzungszeit bekannt und für den Compiler ableitbar. Die Aufrufe von Protokolloperationen können zur Ausführungszeit entfallen.
- **t5** führt 10000 mal eine Leseoperation durch. Im Fall von ΤΕΛΟΣ ist sie nicht-generisch.
- **t6** führt 10000 mal eine Schreiboperation durch. Im Fall von ΤΕΛΟΣ ist sie nicht-generisch.
- **t7** ruft 10000 mal eine nicht-generische Funktion mit *einem* Argument auf. Dieser Test gibt einen Hinweis darauf, wie schnell der generische Dispatch im Vergleich zu regulären Funktionsaufrufen ist.
- **t8** führt 10000 mal den generischen Dispatch über *ein* Argument durch.
- **t9** führt 10000 mal den generischen Dispatch über *zwei* Argumente durch.
- **t5a-d** führen 10000 mal eine generische Leseoperation jeweils mit einer Primär-, zusätzlich mit einer Before-, zusätzlich mit einer After- und schließlich auch einer Around-Methode durch. Hiermit wird die Effizienz der Methodenkombination in Verbindung mir Leseoperationen getestet. Da Leseoperationen in ΤΕΛΟΣ nicht-generisch sind, wurde eine generische Funktion definiert, die die Lesefunktion explizit aufruft.
- **t8a-d** ruft 10000 mal eine generische Funktion jeweils mit einer Primär-, zusätzlich mit einer Before-, zusätzlich mit einer After- und schließlich auch einer Around-Methode durch. Hiermit wird die Effizienz der Methodenkombination für reguläre generische Funktionen getestet.

Alle Tests wurden auf zwei Hardware- und Software-Plattformen durchgeführt: auf einer Sun Ultrasparc Workstation unter Solaris mit Franz Allegro CommonLisp 4.3.1 und auf einem Apple Macintosh PB280 unter MacOS 7.5 mit Macintosh CommonLisp 4.1. Die genauen Daten der Testumgebungen findet man im Anhang D. Die Tests wurden mehrmals durchgeführt und es wurden entsprechende Durchschnittswerte gebildet. Dabei wurden Schwankungen bis zu 30 % vom Durchschnitt festgestellt.

In den folgenden Unterabschnitten stelle ich ausführlich die Ergebnisse der Ausführungszeiten und des Speicherbedarfs von ΤΕΛΟΣ im Vergleich mit CLOS in Franz Allegro CommonLisp 4.3.1 dar, dem derzeit besten kommerziellen CLOS-System. Die entsprechenden Ergebnisse in Macintosh CommonLisp stelle ich nur zusammenfassend dar.

Statische und dynamische Objekterzeugung

Die Abbildungen 8.18 und 8.19 zeigen die Ausführungszeiten und den Speicherbedarf bzgl. der Klassendefinition **t1**, der Methodendefinition **t2**, der dynamischen Instanzerzeugung **t3** und der statischen Instanzerzeugung **t3a**. Bei der Klassen- und Methodendefinition ist ΤΕΛΟΣ rund 5 mal schneller. Auch der Speicherbedarf ist insbesondere bei Cons-Zellen um den Faktor 6 bzw. 30 geringer, was sich bei der Speicherbereinigungszeit positiv bemerkbar macht. Bei der Instanzerzeugung zeigt sich ein großer Unterschied: statische Aufrufe von `make-instance` werden in CLOS so optimiert, daß sie 35 mal schneller sind als dynamische Aufrufe. Dies zeigt, daß das spezifizierte Protokoll für eine dynamische Sprache nicht angemessen ist. Im dynamisch Fall ist ΤΕΛΟΣ rund 7 mal schneller als CLOS. In meiner Implementierung von ΤΕΛΟΣ gibt es keinen Unterschied zwischen dynamischen und statischen Aufrufen von `make`. Ein Modul- oder Applikations-Compiler kann aber für ΤΕΛΟΣ mindestens die gleichen Optimierungen durchführen wie für CLOS. Diese Meßergebnisse bestätigen sehr eindrucksvoll, wie wichtig statische Analysen und Optimierungen zur Übersetzungszeit sind. ΤΕΛΟΣ ist systematisch darauf ausgerichtet, eine hohe Effizienz für statische und dynamische Fälle zu erreichen, während CLOS systematisch auf dynamische Implementierungstechniken ausgerichtet ist und die statische Instanzerzeugung eher eine Ausnahme darstellt.

Feldzugriffe

Die Abbildungen 8.20 und 8.21 zeigen die Ausführungszeiten und den Speicherbedarf bzgl. der Feldzugriffe, aufgeteilt in dynamische (**t4**) und statische (**t4a**) Aufrufe von `slot-value` sowie Aufrufe der Lese- (**t5**) und Schreiboperationen (**t6**). In **t4** wird `slot-value` direkt aufgerufen, in **t4a** innerhalb einer Methode. Der Wert von **t4a** enthält daher zusätzlich den Aufruf einer generischen Funktion. Offensichtlich werden statische `slot-value`-Aufrufe in CLOS so optimiert, daß sie im Vergleich zum Aufruf einer generische Funktion kaum ins Gewicht fallen. ΤΕΛΟΣ stellt aus konzeptionellen Gründen ein generelles Primitiv wie `slot-value` nicht zur Verfügung. Des Vergleichs mit CLOS wegen habe ich es hier mit den vorhandenen Protokolloperationen implementiert, ohne besondere Optimierungen vorzunehmen. Daher ist der Wert im statischen Fall auch höher als im dynamischen Fall (es kommt ja der Aufruf einer generische Funktion hinzu).

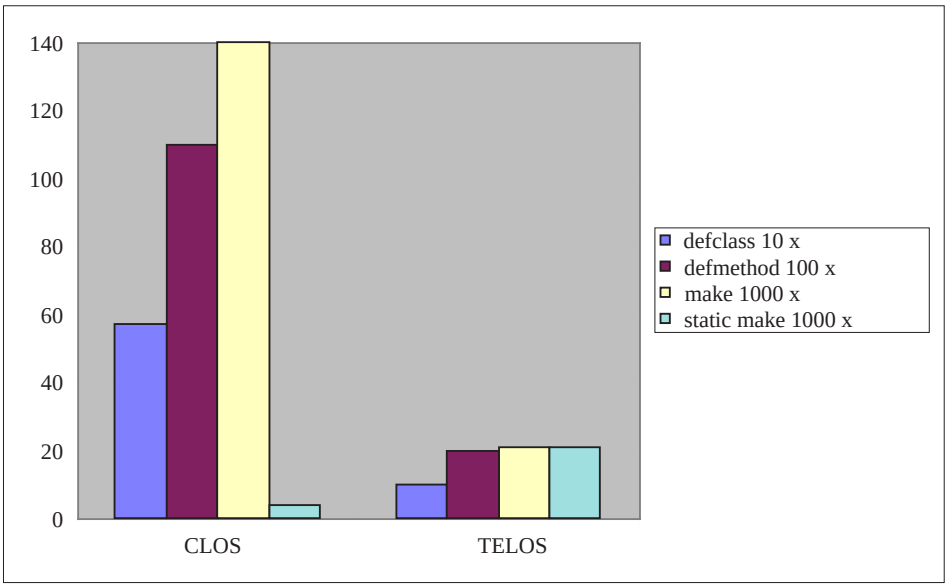


Abbildung 8.18: Statische und dynamische Objekterzeugung **t1**, **t2**, **t3** und **t3a**: Vergleich der Ausführungszeiten von CLOS und ΤΕΛΟΣ auf einer Sun Workstation in Franz Allegro CL 4.3.1 in Millisekunden.

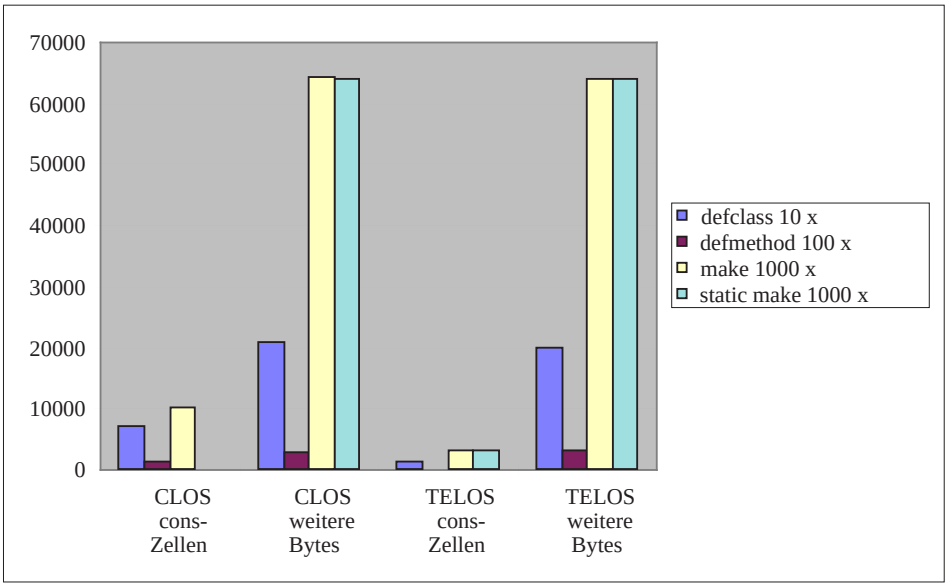


Abbildung 8.19: Statische und dynamische Objekterzeugung **t1**, **t2**, **t3** und **t3a**: Vergleich des Speicherbedarfs von CLOS und ΤΕΛΟΣ in Franz Allegro CL 4.3 auf Sun Workstation in Cons-Zellen und Bytes.

Wesentlich wichtiger ist der Vergleich der Lese- und Schreiboperationen. Da diese in TELOS nicht-generisch sind, sind sie etwas effizienter als die hochoptimierten generischen Lese- und Schreiboperationen von CLOS. Ein TELOS-Modul- bzw. Applikations-Compiler kann natürlich wesentlich bessere Werte erreichen als meine portable Implementierung in COMMONLISP. Bemerkenswert ist hier, daß die CLOS-Optimierungen offensichtlich Speicherplatz kosten, während TELOS so gut wie keinen dynamischen Speicher verbraucht. TELOS wird daher besser im Gesamtverhalten komplexer Systeme abschneiden.

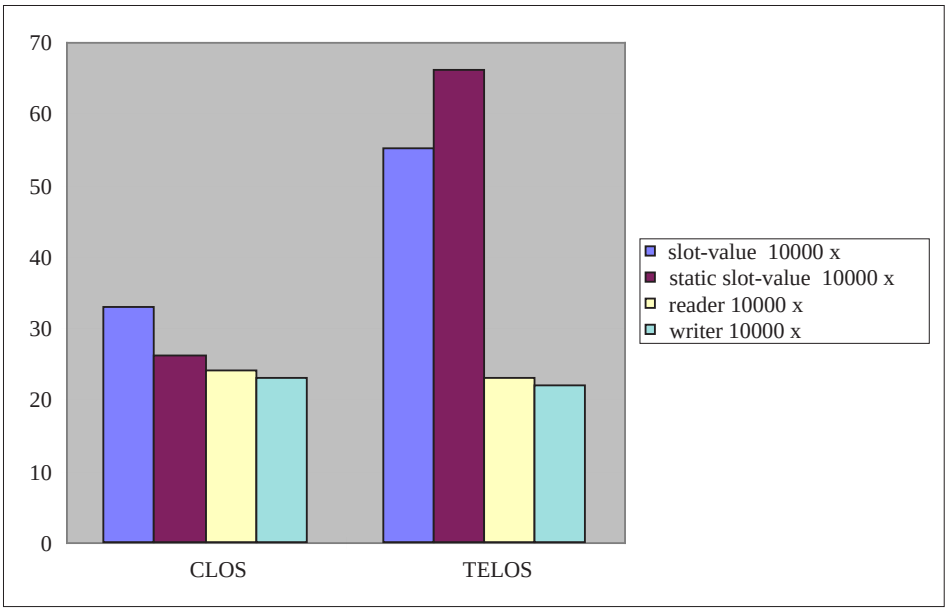


Abbildung 8.20: Statische und dynamische Feldzugriffe **t4**, **t4a**, **t5** und **t6**: Vergleich der Ausführungszeiten von CLOS und TELOS in Franz Allegro CL 4.3 auf Sun Workstation in Millisekunden.

Generischer Dispatch

Die Abbildungen 8.22 und 8.23 zeigen die Ausführungszeiten und den Speicherbedarf bzgl. generischer Funktionsaufrufe mit einem (**t8**) und mit zwei (**t9**) Dispatch-Argumenten. **t7** zeigt zum Vergleich den Wert für eine nicht-generische Funktion mit einem Argument. Daran sieht man, daß der generische Dispatch in Franz CLOS schon stark optimiert wurde. Ein Modul- oder Applikations-Compiler kann für TELOS mindestens die gleichen Werte erreichen. Aber auch meine portable Implementierung in COMMONLISP ist nur ca. 20 bis 25 % langsamer als Franz CLOS. Offensichtlich gehen die CLOS-Optimierungen auf Kosten des dynamischen Speicherbedarfs, der 4 bis 9 mal höher ist als bei TELOS.

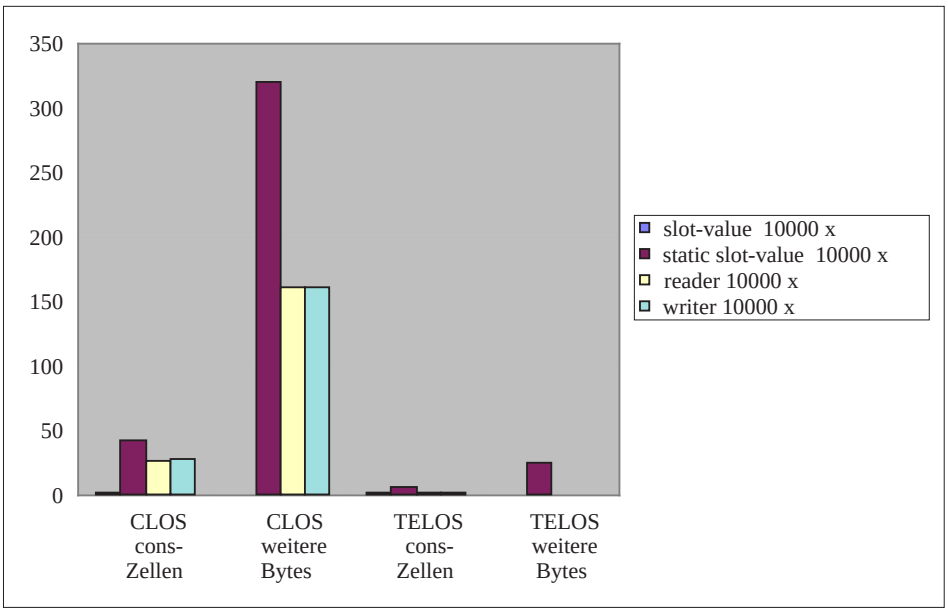


Abbildung 8.21: Statische und dynamische Feldzugriffe **t4**, **t4a**, **t5** und **t6**: Vergleich des Speicherbedarfs von CLOS und TELOS in Franz Allegro CL 4.3 auf Sun Workstation in Cons-Zellen und Bytes.

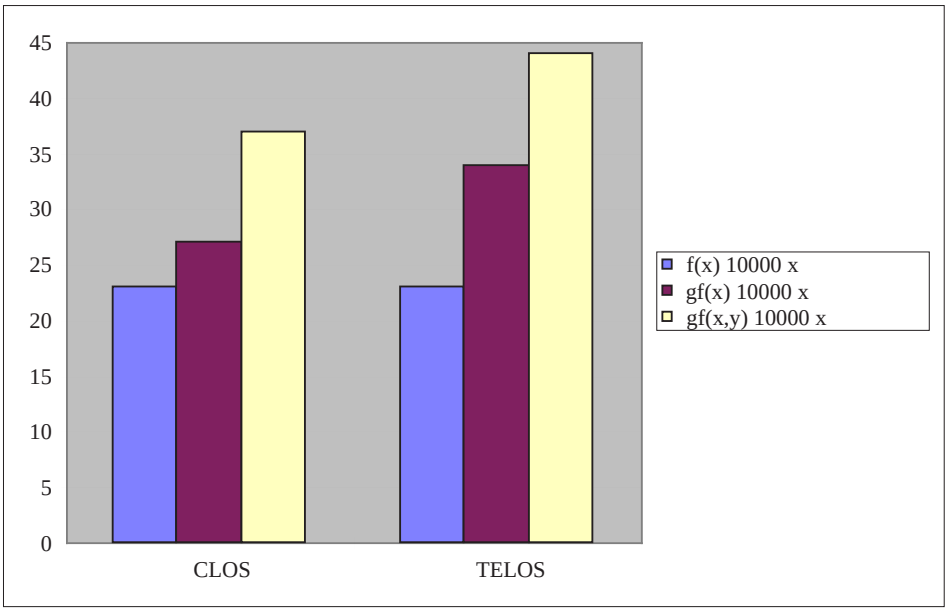


Abbildung 8.22: Generischer Dispatch **t7**, **t8**, **t9**: Vergleich der Ausführungszeiten von CLOS und TELOS in Franz Allegro CL 4.3 auf Sun Workstation in Millisekunden.

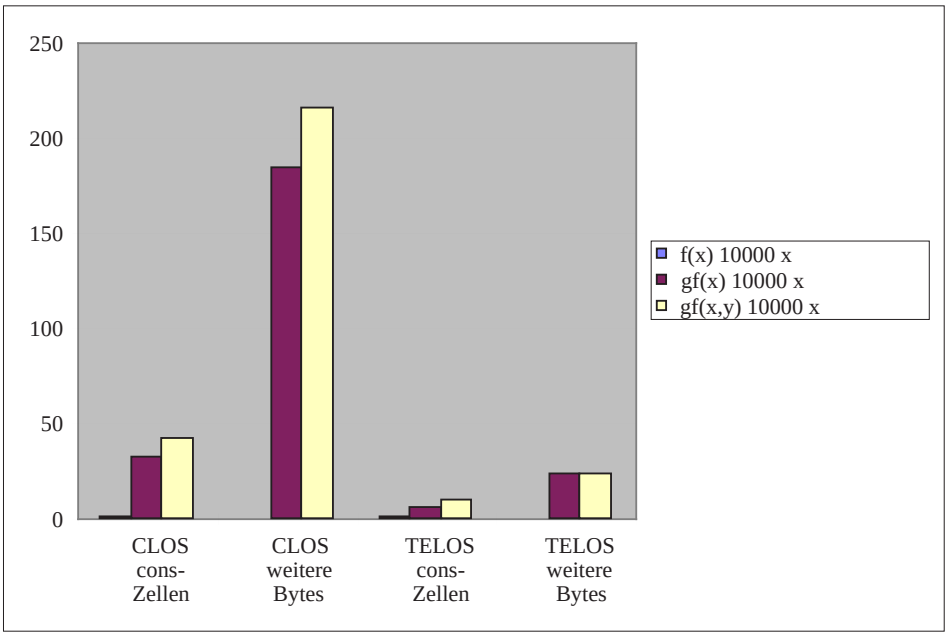


Abbildung 8.23: Generischer Dispatch **t7**, **t8**, **t9**: Vergleich des Speicherbedarfs von CLOS und ΤΕΛΟΣ in Franz Allegro CL 4.3 auf Sun Workstation in Cons-Zellen und Bytes.

Methodenkombination

Die Abbildungen 8.24 und 8.25 zeigen die Ausführungszeiten und den Speicherbedarf bzgl. der Methodenkombination. Test **t8a** ist identisch mit **t8**. Bei den Tests **t8b**, **t8c** und **t8d** wird die Standardmethodenkombination von CLOS gemessen. Da TELOS im Kern nur einfache Methodenkombination bereitstellt, werden hier Before-, After- und Around-Methoden über Primärmethoden und `call-next-method` simuliert (siehe auch Abschnitt 8.3.4). Es fällt auf, daß die Kombinationszeit mit Before- und After-Methoden in CLOS stark optimiert wurde. Bei Around-Methoden ist meine portable TELOS-Implementierung genau so schnell wie Franz CLOS. Allerdings sind die CLOS-Laufzeitoptimierungen der Methodenkombination mit enormem Speicherverbrauch verbunden: schon bei Before- und After-Methoden wird etwa 20 bis 40 mal mehr Speicher dynamisch verbraucht als in TELOS. Und mit Around-Methoden springt der Speicherbedarf ins Unermeßliche: hier braucht CLOS etwa 4000 mal mehr Cons-Zellen und rund 100000 mal mehr Bytes als TELOS. Spätestens hier wird deutlich, daß im Sprachdesign von CLOS die Kriterien Einfachheit und Effizienz nicht ausreichend berücksichtigt wurden.

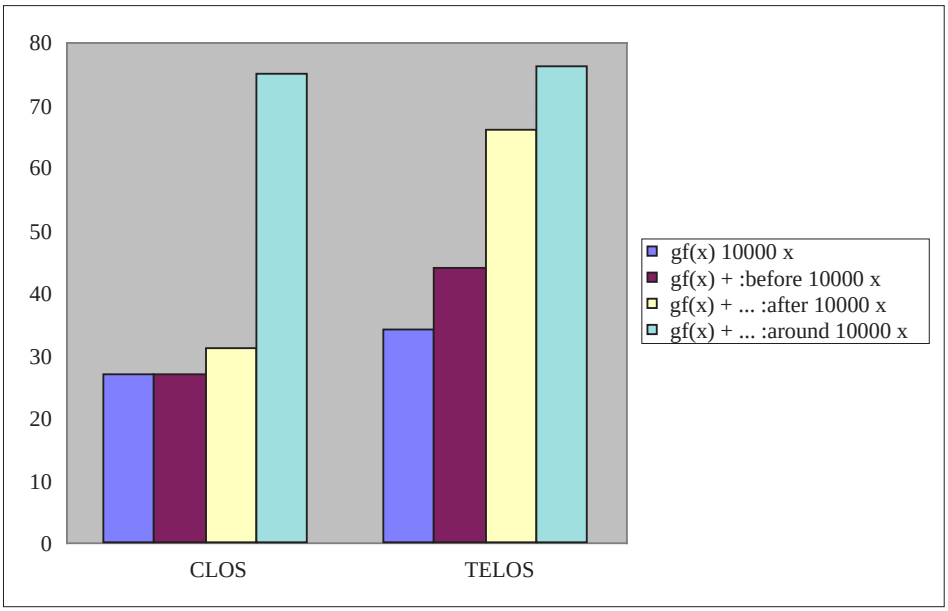


Abbildung 8.24: Methodenkombination **t8a-d**: Vergleich der Ausführungszeiten von CLOS und TELOS in Franz Allegro CL 4.3 auf Sun Workstation in Millisekunden.

Summe aller Tests in Franz Allegro Common Lisp 4.3.1

Die Abbildungen 8.26 und 8.27 zeigen die Ausführungszeiten und den Speicherbedarf der obigen Tests insgesamt. Wie schon oben gesagt, mag man über die angemessene Gewichtung der Tests untereinander streiten. Beispielsweise kann man die statische Instanzerzeugung sowie den generischen Dispatch höher gewichten, so daß sich das Gesamtergebnis für CLOS verbessert. Aber dies will ich ja gar nicht bestreiten: Compileroptimierungen auf

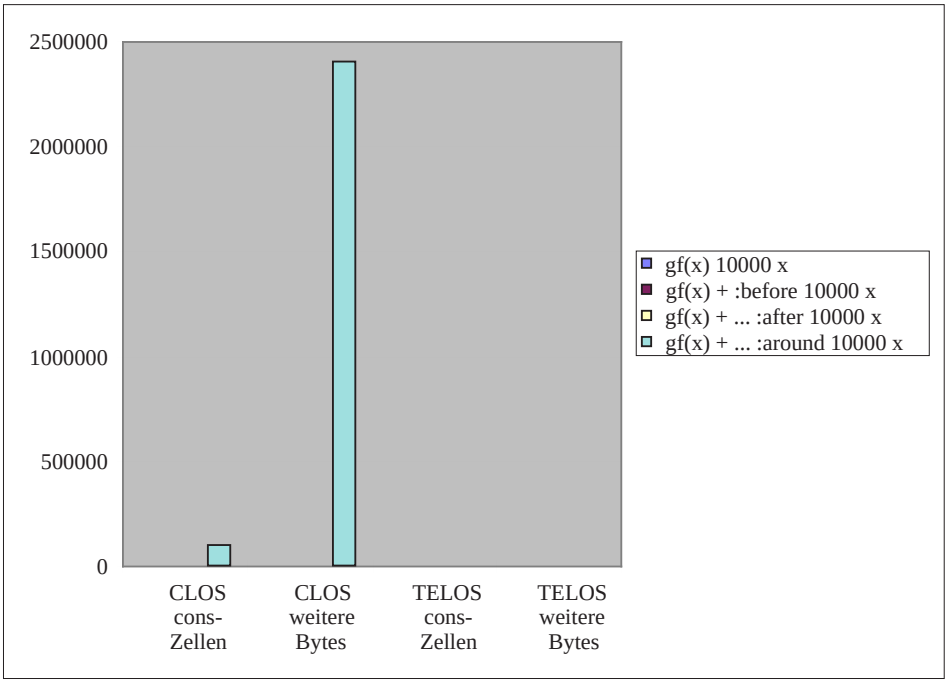


Abbildung 8.25: Methodenkombination **t8a-d**: Vergleich des Speicherbedarfs von CLOS und ΤΕΛΟΣ in Franz Allegro CL 4.3 auf Sun Workstation in Cons-Zellen und Bytes.

Grund von statischen Analysen sind den dynamischen Einbettungstechniken überlegen. Ein ΤΕΛΟΣ-Compiler würde hierbei nicht schlechter, sondern besser abschneiden, wenn schon die portable Implementierung diese Resultate zeigt. Dies um so mehr, wenn man bedenkt, daß Franz CommonLisp die derzeit beste kommerzielle Implementierung von CLOS anbietet.

Faßt man den Performanzvergleich von ΤΕΛΟΣ mit CLOS am Beispiel meiner portablen Referenzimplementierung und Franz CommonLisp 4.3.1 auf einer Sun Workstation zusammen, so ist ΤΕΛΟΣ insgesamt eindeutig effizienter. Bis auf die statische Instanzerzeugung werden die leichten Laufzeitvorteile von CLOS durch einen unverhältnismäßig hohen Speicherbedarf erkauft.

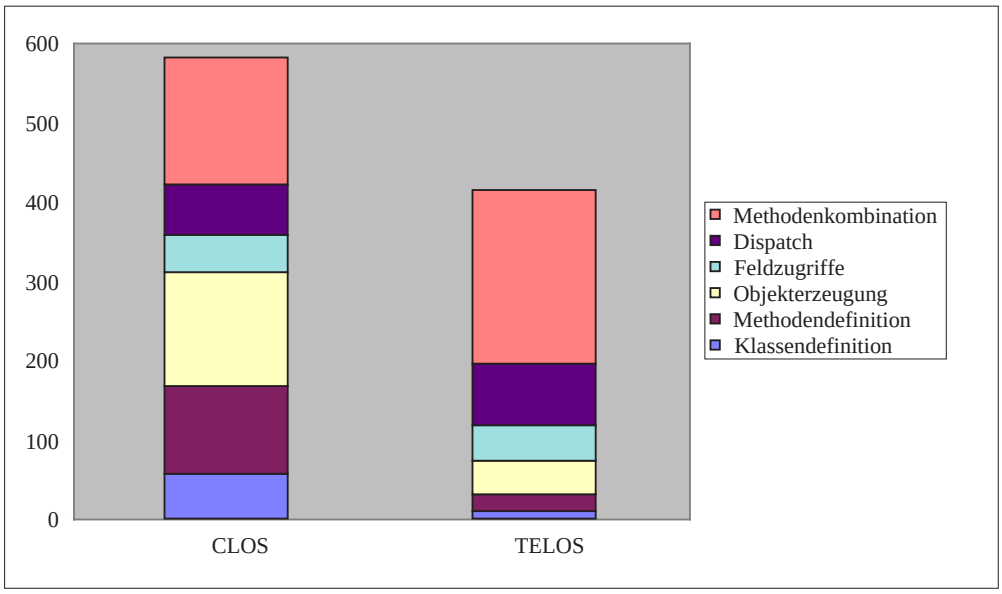


Abbildung 8.26: Vergleich der Gesamtausführungszeiten von **t1**, **t2**, **t3**, **t3a**, **t5**, **t6**, **t8**, **t9** und **t8a-d** für CLOS und ΤΕΛΟΣ in Franz Allegro CL 4.3 auf Sun Workstation in Millisekunden.

Ergebnisse für Macintosh Common Lisp 4.1

Alle Performanzmessungen wurden auch auf einem Apple PB280 in Macintosh Common Lisp 4.1 durchgeführt. Dabei wurden zusätzlich die Systeme MCS 0.5 und MCS 1.0 getestet.[6]

Bis auf den generischen Dispatch bei einem Argument sowie die Feldzugriffe über `slot-value` schneidet ΤΕΛΟΣ überall besser ab als CLOS. Der Speicherbedarf von ΤΕΛΟΣ ist

[6] Als weitere Implementierung von ΤΕΛΟΣ, ebenfalls in CommonLisp, habe ich auch das System von Russel Bradford [Bradford, 1996] getestet. Da es aber nur unter dem Aspekt der Validierung der ΤΕΛΟΣ-Spezifikation realisiert wurde, ist es nicht effizient genug, um hier aufgeführt zu werden. Die Ergebnisse findet man im Anhang D.2.

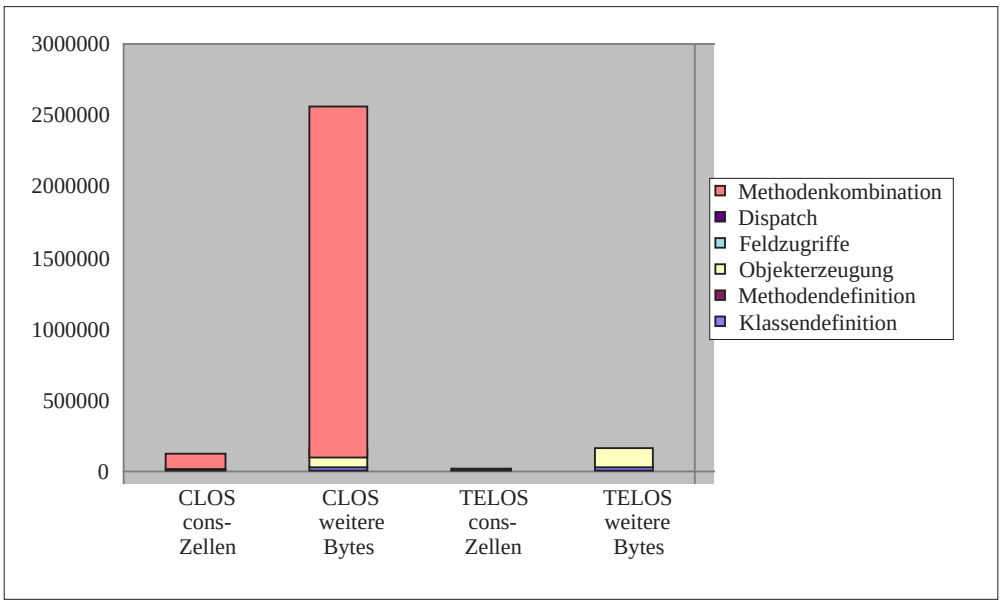


Abbildung 8.27: Vergleich des Gesamtspeicherbedarfs von **t1**, **t2**, **t3**, **t3a**, **t5**, **t6**, **t8**, **t9** und **t8a-d** für CLOS und ΤΕΛΟΣ in Franz Allegro CL 4.3 auf Sun Workstation in Cons-Zellen und Bytes.

nur bei der Instanzerzeugung höher als von CLOS. Hier wirken sich die Nachteile einer portablen Implementierung aus.

Interessant ist, daß selbst MCS 0.5 in der Summe besser abschneidet als CLOS. Dies liegt im wesentlichen an der schnelleren Methodendefinition und an der schnelleren Instanzerzeugung. Letztere ist sogar schneller als in ΤΕΛΟΣ, dank der inkrementellen Berechnung effizienter Konstruktoren. Ihr Nachteil war jedoch, daß der inkrementelle Compiler während der Klassendefinition explizit aufgerufen werden mußte. Deutlich höher ist der dynamische Speicherbedarf bei MCS 0.5 und MCS 1.0, was zum Teil am Sprachdesign, aber auch an den gewählten Implementierungstechniken liegt[7].

Vergleicht man die beiden CLOS-Implementierungen miteinander, so zeigt sich, daß die Prioritäten bei den Optimierungen unterschiedlich gesetzt wurden. Das Macintosh CommonLisp CLOS ist in der Laufzeit nicht so effizient, geht aber wesentlich sparsamer mit dem Speicherplatz um[8].

Als Fazit des Performanzvergleichs auf beiden Plattformen, Sun und Apple, halte ich fest, daß die Effizienz der einfachen portablen Implementierung von ΤΕΛΟΣ in CommonLisp das ausgewogene Sprachdesign von ΤΕΛΟΣ klar bestätigt. Eine weitere Bestätigung liefert die EuLisp-Implementierung YOUTOO [Kind, 1998]. YOUTOO stellt einen Bytecode-Compiler und -Interpretierer zur Verfügung, der effizienter als alle vergleichbaren Systeme ist und selbst im Vergleich mit dem Maschinencode-Compiler in Franz Allegro CommonLisp nur um den Faktor 2 langsamer ist.

8.3 Exemplarische Spracherweiterungen

Die Erweiterbarkeit von ΤΕΛΟΣ stellt ein wichtiges Designkriterium dar. Um das Resultat besser einschätzen zu können, stelle ich einige typische Erweiterungen auf konzeptueller Ebene exemplarisch dar. Die wichtigste davon ist die Mixin-Vererbung. Sie wird am ausführlichsten behandelt, bis hin zur vollständigen Implementierung im Anhang E. Hinzu kommen Erweiterungen für das Redefinieren von Klassen mit automatischer Anpassung aller abhängigen Klassen und Instanzen sowie für das dynamische Reklassifizieren von Instanzen, die sogenannte *Change-Class*-Funktionalität. Abschließend gehe ich kurz auf eine größere ΤΕΛΟΣ-Anwendung aus der Künstlichen Intelligenz ein: Der Wissensrepräsentationsformalismus TINA stellt Konzepte und Relationen als Modellierungsmittel zur Verfügung. Diese werden in ΤΕΛΟΣ harmonisch integriert und als Spezialisierung des Metaobjektprotokolls realisiert [Kopp, 1996a].

8.3.1 Mixin-Vererbung

Das Konzept der Mixin-Vererbung wurde bereits in 4.3.3 ausführlich diskutiert. In MCS 1.0 wurde sie schon im Objektsystemkern bereitgestellt und benutzt, siehe Seite 203. In

[7] Der Wert von **t9** wurde für MCS 0.5 als die doppelte Zeit des einfachen Dispatchs berechnet, weil MCS 0.5 keinen mehrfachen Dispatch unterstützt.

[8] Eine Folge der erst spät und immer noch nicht zufriedenstellend eingeführten virtuellen Speicherverwaltung im Betriebssystem MacOS.

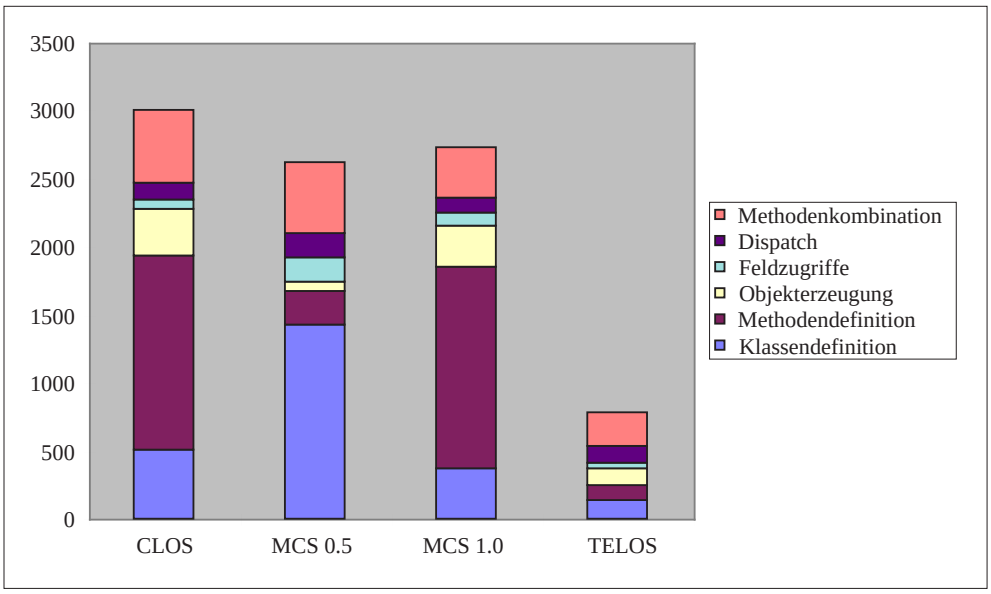


Abbildung 8.28: Vergleich der Gesamtausführungszeiten von **t1**, **t2**, **t3**, **t3a**, **t5**, **t6**, **t8**, **t9** und **t8a-d** für CLOS, MCS 0.5, MCS 1.0 und ΤΕΛΟΣ in Macintosh CL 4.1 auf einem Apple Macintosh PB280 in Millisekunden.

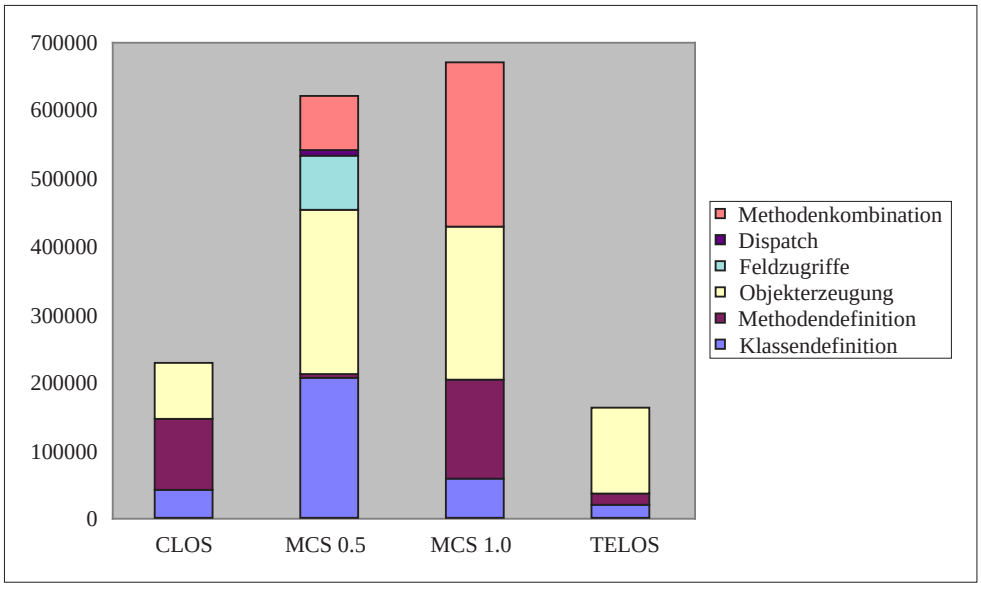


Abbildung 8.29: Vergleich des Gesamtspeicherbedarfs von **t1**, **t2**, **t3**, **t3a**, **t5**, **t6**, **t8**, **t9** und **t8a-d** für CLOS und ΤΕΛΟΣ in Macintosh CL 4.1 auf einem Apple Macintosh PB280 in Bytes.

diesem Abschnitt wird sie unter dem Gesichtspunkt ihrer Unterstützung als Spracherweiterung präsentiert. Dabei verwende ich zunächst die Notation aus Kapitel 4, deren Umsetzung in TELOS aber keinerlei Probleme bereitet.

Das Ziel der Mixin-Vererbung besteht darin, die Beschränkungen der einfachen Vererbung soweit zu öffnen, daß man die Nachteile der einfachen Vererbung überwindet, sich aber nicht gleichzeitig alle Nachteile der allgemeinen multiplen Vererbung einhandelt. Anknüpfend an Organisationsprinzipien von Begriffsontologien natürlicher Sprachen werden bei der Klassifikation von Objekten essentielle bzw. dingliche von additiven Eigenschaften unterschieden. Entsprechend unterscheidet man Basisklassen von Mixin-Klassen:

- Eine Basisklasse kann als direkte Subklasse mehrerer Mixin-Klassen aber nur einer anderen Basisklasse definiert werden.
- Eine Mixin-Klasse kann als direkte Subklasse mehrerer Mixin-Klassen definiert werden, aber nicht als Subklasse einer Basisklasse.
- Sogenannte Join-Klassen sind in Vererbungshierarchien nicht zulässig, außer der Klasse `Object`, die Oberklasse aller Klassen ist.
- Basisklassen können direkt instanziierbar sein. Alle Mixin-Klassen sind abstrakt, d. h. sie werden nie direkt instanziiert.

Der Objektsystem-Kern stellt die Klassen `Class` und `SimpleClass` zur Verfügung. Zunächst blieb noch offen, ob beide vom Benutzer spezialisierbar sein sollen, zumindest ist aber `Class` spezialisierbar. Die Erweiterung des Kerns um das Konzept der Mixin-Vererbung besteht nun darin, zwei weitere Klassen mit entsprechenden Methoden für die betroffenen generischen Operationen des Metaobjektprotokolls zu definieren. Die Klassenhierarchie wird um die Klassen `BaseClass` und `MixinClass` als Subklassen von `Class` erweitert:

```
Object
  MetaObject
    Class
      SimpleClass
      BaseClass
      MixinClass
    Field
      SimpleField
    GenericFunction
      SimpleGenericFunction
    Method
      SimpleMethod
```

Falls `SimpleClass` spezialisierbar wäre, könnte `BaseClass` auch Subklasse von `SimpleClass` sein. Die Entscheidung darüber hängt nicht zuletzt davon ab, welche Optimierungen im Objektsystem-Kern tatsächlich vorgenommen werden und ob die entsprechenden Voraussetzungen auch für Basisklassen zutreffen. Diese Voraussetzungen hängen im wesentlichen davon ab, ob eine Klasse einfach oder mehrfach von anderen Klassen spezialisiert

```
compatibleSuperclasses? cl directSuperclasses → boolean
compatibleSuperclass? cl superclass → boolean
computeCPL cl directSuperclasses → (cl*)
computeInheritedKeywords cl directSuperclasses → ((keyword*)*)
computeKeywords cl directKeywords inheritedKeywords → (keyword*)
computeInheritedFields cl directSuperclasses → ((field*)*)
computeFields cl fieldSpecs inheritedFields → (field*)
computeNewField cl fieldSpec → field
computeNewFieldClass cl fieldSpec → fieldClass
computeSpecializedField cl inheritedFields fieldSpec → field
computeSpecializedFieldClass cl inheritedFields fieldSpec → fieldClass
computeInstanceSize cl fields → integer
computeAndEnsureFieldAccessors cl fields inheritedFields → (field*)
computeFieldReader cl field fields → function
computeFieldWriter cl field fields → function
ensureFieldReader cl field fields reader → function
ensureFieldWriter cl field fields writer → function
computePrimitiveReaderUsingField field cl fields → function
computePrimitiveWriterUsingField field cl fields → function
computePrimitiveReaderUsingClass cl field fields → function
computePrimitiveWriterUsingClass cl field fields → function
```

Abbildung 8.30: Potentiell zu spezialisierende Operationen des Metaobjektprotokolls

(ererbt) werden darf. Somit könnte `BaseClass` diesbezüglich wie `SimpleClass` behandelt werden. Insbesondere können die in Basisklassen definierte Felder effizienter zugegriffen werden (siehe auch [Shalit, 1996, S. 134]).

Nun muß man prüfen, welche generischen Operationen des Metaobjektprotokolls spezialisiert werden müssen und welche unverändert geerbt werden können. Zu prüfen sind die in Abbildung 8.30 aufgeführten generischen Operationen.

Berücksichtigt man die von der Mixin-Erweiterung gestellten Anforderungen bereits im Objektsystem-Kern, so kann sich die tatsächliche Spezialisierungsaufgabe deutlich vereinfachen. Dazu wird das Verhalten des Kerns etwas genereller spezifiziert, als für einfache Vererbung notwendig. Dies ist jedoch durchaus akzeptabel, da das gesamte Protokoll schon darauf zielt, verschiedene Vererbungsstrategien als Erweiterung zu unterstützen. Natürlich sollte die Effizienz des Objektsystem-Kerns nicht darunter leiden. Von folgenden Operationen kann angenommen werden, daß sie hinreichend generell sind:

```
computeInheritedKeywords
computeKeywords
computeInheritedFields
computeFields
computeNewField
computeNewFieldClass
computeSpecializedField
computeSpecializedFieldClass
```

```
compatibleSuperclasses? cl directSuperclasses → boolean
compatibleSuperclass? cl superclass → boolean
computeCPL cl directSuperclasses → (cl*)
computeFieldReader cl field fields → function
computeFieldWriter cl field fields → function
ensureFieldReader cl field fields reader → function
ensureFieldWriter cl field fields writer → function
```

Abbildung 8.31: Tatsächlich zu spezialisierende Operationen des Metaobjektprotokolls

```
computeInstanceSize
computeAndEnsureFieldAccessors
computePrimitiveReaderUsingField
computePrimitiveWriterUsingField
computePrimitiveReaderUsingClass
computePrimitiveWriterUsingClass
```

Zu spezialisieren sind dann nur noch die generischen Operationen in Abbildung 8.31[9]. Deutlich höher wäre der Aufwand, wenn es mehrere Felder mit demselben Namen zu erben gäbe. Dies entfällt jedoch, da ich bei der Diskussion des Mixin-Vererbungskonzepts folgende Entscheidungen getroffen habe:

1. Die Feldnamen unterliegen dem Import und Export auf Modulebene.
2. Alle Felder bzw. Feldnamen von Klassen, die in keiner Super- bzw. Subklassenrelation stehen, sind verschieden.
3. Alle Superklassen einer zu definierenden Klasse (direkte und indirekte) müssen bis auf `Object` disjunkt sein. Es gibt keine anderen Join-Knoten in der Vererbungshierarchie.

Diese Betrachtung sollte deutlich machen, daß die Designentscheidungen nicht willkürlich getroffen wurden, sondern daß wie bei einem Puzzle plötzlich mehrere Vereinfachungen eintreffen. Entsprechende Methoden für `compatibleSuperclasses?`, `compatibleSuperclass?` und `computeCPL` der Klassen `BaseClass` und `MixinClass` müssen die postulierten Beschränkungen bzgl. der direkten Superklassen ihrer Instanzen (die Klassenobjekte sind) überprüfen und ggf. Fehler melden. Die Operationen des Feldzugriffsprotokolls `computeFieldReader`, `computeFieldWriter`, `ensureFieldReader` und `ensureFieldWriter` müssen spezialisiert werden, weil die Zugriffsoperationen für Mixin-Felder generisch sein sollten (siehe 4.4.5).

Eine vollständige Implementierung der Mixin-Vererbung als Erweiterung des Objektsystemkerns von TELOS befindet sich im Anhang E.

[9] In CELOS, siehe Anhang E, sind es mehr, weil in den bereitgestellten Methoden des Objektsystemkerns die Annahme einfacher Vererbung getroffen wurde. Dies kann leicht geändert werden, indem ein Teil der Methoden aus dem Mixins-Modul in den Objektsystemkern übernommen wird.

8.3.2 Redefinieren von Klassen

Aus der Sicht des Programmiersprachen-Designers gehört das Redefinieren von Klassen eher zu den fragwürdigen Konzepten. Insbesondere dann, wenn man eine einfache Semantik sowie robuste Compiler und Laufzeitsysteme anstrebt. Und natürlich erst recht, wenn man an ihre Verifikation denkt [Goerigk, 1993], [Goerigk, 1997]. Auf der anderen Seite ist für die meisten höheren Programmiersprachen die inkrementelle und interaktive Programmentwicklung mittlerweile zur Regel geworden. Dies galt auch schon immer für die Wissensrepräsentationssprachen. Für die Modellierung von Anwendungsdomänen braucht man noch flexiblere Sprachkonstrukte und Entwicklungswerkzeuge. Ihre Implementierung kann drastisch verkürzt werden, wenn man sie als Erweiterung der Programmentwicklungswerkzeuge einer erweiterbaren objektorientierten Programmiersprache konzipieren kann. Hier wird exemplarisch gezeigt, wie die Funktionalität des Redefinierens von Klassen mit Hilfe des Metaobjektprotokolls unterstützt werden kann. Dabei greife ich auf die Erfahrungen aus MCS 1.0 zurück, siehe Seite 208 sowie [Bretthauer und Kopp, 1991].

Das Hauptproblem, das sich im Zusammenhang mit dem Redefinieren von Klassen stellt, ist die Abhängigkeitsverwaltung. Wird eine Klasse redefiniert, so müssen alle von ihr abhängigen Objekte entsprechend an die neue Definition angepaßt werden. Dies sind alle direkten und indirekten Subklassen sowie deren möglicherweise bereits erzeugten Instanzen. Die Abhängigkeiten sind um so größer, je mehr Optimierungen, wie z. B. Inline-Kompilation von Lese- und Schreiboperationen, vom System vorgenommen werden.

Um die genannten Abhängigkeiten zu verwalten, müssen zusätzliche Informationen mit einem Klassenobjekt während seiner Initialisierung assoziiert werden:

- die direkten Superklassen,
- die direkten Subklassen und
- die Initialisierungsargumente.

Nicht notwendig ist es im Unterschied zu CLOS, die sogenannten direkten Methoden mit einer Klasse zu assoziieren. Dies sind Methoden, in deren Bereich die Klasse explizit vorkommt.

Die Grundidee bei der Abhängigkeitsverwaltung besteht darin, den Aufwand für Anpassungen dadurch zu minimieren, daß man sie nur bei Bedarf vornimmt. Wird eine Klasse K redefiniert, so müssen folgende Aktionen durchgeführt werden:

1. Entferne K als direkte Subklasse ihrer direkten Superklassen.
2. Markiere K als nicht initialisiert.
3. Markiere alle Subklassen von K als nicht initialisiert.
4. Markiere alle Instanzen von K und die ihrer Subklassen als obsolet.
5. Reinitialisiere K gemäß der neuen Definition.

In diesem Zusammenhang ist dafür zu sorgen, daß Instanzen redefinierbarer Klassen weiterhin der automatischen Speicherbereinigung unterliegen. Daher ist es nicht möglich, in den Klassen Referenzen auf ihre Instanzen zu halten. Der CLOS-Ansatz besteht darin, daß Instanzen indirekte Referenzen auf ihre Klasse besitzen, die *Class-Wrapper* genannt werden. Jede Klasse besitzt einen Class-Wrapper, der ihrer aktuellen Definition entspricht und möglicherweise mehrere alte Class-Wrapper, die den früheren Definitionen entsprechen. Versucht man auf einer noch nicht angepaßten Instanz eine Operation auszuführen, stellt CLOS anhand des markierten Class-Wrappers fest, daß die Instanz erst angepaßt werden muß. Mit der Anpassung erhält die Instanz einen aktuellen Class-Wrapper. In CLOS sind bereits alle vordefinierten Operationen auf das Redefinieren von Klassen vorbereitet.

Mein Ziel ist es, den Kern möglichst klein und effizient zu halten. Dadurch wird die Unterstützung des Redefinierens als Erweiterung zunächst etwas schwieriger, als wenn man es von vornherein für alle Objekte vorsieht. Auf der anderen Seite helfen einige Designverbesserungen bei den Zugriffsoperationen wieder für eine Vereinfachung. Da alle Feldzugriffe über eindeutige Zugriffsoperationen erfolgen, brauchen nur diese um entsprechende Überprüfung auf Aktualität erweitert zu werden. Das heißt, daß obsolete Instanzen in meinem Ansatz spätestens bei einem Feldzugriff aktualisiert werden. Der Overhead für das Redefinieren und das automatische Anpassen abhängiger Objekte ist somit nicht höher als in CLOS, ist jedoch nur für Objekte der Erweiterung zu erbringen. Wird diese aufwendige Funktionalität nicht gebraucht, muß für sie auch kein Preis bezahlt werden. Dies ist ein wichtiges Resultat meiner Arbeit.

Die Erweiterung des Kerns um das Konzept des Redefinierens von Klassen besteht nun darin, zwei weitere Klassen mit entsprechenden Methoden für die betroffenen generischen Operationen des Metaobjektprotokolls zu definieren. Die Klassenhierarchie wird um die Klasse `RedefinableClass` als Subklasse von `Class` und um `UpdatableObject` als Subklasse von `Object` erweitert:

```
Object
  UpdatableObject
  MetaObject
    Class
      SimpleClass
      RedefinableClass
```

Redefinierbare Klassen werden als Instanzen von `RedefinableClass` und als Subklassen von `UpdatableObject` erzeugt. Die generischen Operationen des Allokationsprotokolls müssen so spezialisiert werden, daß anpaßbare Instanzen ihre Struktur unter Beibehaltung der Identität ändern können. Die einfache und naheliegende Lösung besteht in der zweistufigen Speicher-Repräsentation. Die eigentlichen Felder eines Objekts werden in einem einfachen Array fester Länge abgelegt. Jedes Objekt bekommt neben dem internen Verweis auf seine Klasse auch einen internen Verweis auf sein Array der Feldwerte. Ändert sich nun die Struktur einer Klasse, so muß nur dieses Array durch ein neues ersetzt werden. Wie in CLOS kann man Class-Wrapper verwenden, um die Aktualität von Objekten zu überprüfen. Dazu wäre ein weiterer interner Verweis auf den Class-Wrapper notwendig.

Darüberhinaus müssen die generischen Operationen `computePrimitiveReaderUsingClass` sowie `computePrimitiveWriterUsingClass` entsprechend spezialisiert werden. Die

von ihnen berechneten Zugriffsoperationen müssen die Aktualität eines Objekts prüfen und indirekt auf die Feldwerte zugreifen. Natürlich können hier auch ausgeklügelte Implementierungen realisiert werden. Mir geht es darum, eine einfache, robuste und effiziente Lösung anzubieten.

8.3.3 Dynamisches Klassifizieren von Objekten

Die bisherigen Konzepte zur Unterstützung des Klassifizierens von Objekten waren statisch. Das heißt, daß die Klassifikation einer Domäne mit der Erstellung eines Programms vorgegeben und vor Programmausführung abgeschlossen sein mußte. Die Klassenzugehörigkeit von Objekten ist über ihre Lebensdauer fest. Als ersten Schritt in Richtung des dynamischen Klassifizierens bietet CLOS die `change-class`-Funktion an. Dadurch wird ein Sprachkonstrukt zur Verfügung gestellt, mit dem man generelle und anwendungsspezifische dynamische Klassifikationsverfahren realisieren kann. Hierzu zählen insbesondere sogenannte *Classifier* aus Konzeptsprachen der Wissensrepräsentation[10], die auf KL-ONE zurückgehen [Brachman und Schmolze, 1985], [Kopp, 1996a, S. 57, 116]. Natürlich können Klassifikationsverfahren auch ohne die objektorientierten Sprachmittel einer Programmiersprache implementiert werden. Der Aufwand ist dann aber deutlich höher. Wie schon das Redefinieren von Klassen gehört `change-class` zur Standard-Funktionalität in CLOS. Dementsprechend kann im allgemeinen für kein Objekt garantiert werden, daß es seine Klassenzugehörigkeit beibehält. Dies widerspricht meinen Designkriterien. Solche Sprachmittel sollten nur gezielt verwendet werden und nur da zu einem Overhead führen, wo sie wirklich benötigt werden.

Die Erweiterung des Kerns um das Konzept des dynamischen Klassifizierens besteht darin, zwei weitere Klassen mit entsprechenden Methoden für die betroffenen generischen Operationen des Metaobjektprotokolls zu definieren. Die Klassenhierarchie wird um die Klasse `DynamicClass` als Subklasse von `Class` und um `DynamicObject` als Subklasse von `Object` erweitert:

```
Object
  DynamicObject
  MetaObject
    Class
      SimpleClass
      DynamicClass
```

Die Spezialisierung der generischen Operationen hängt von der gewünschten Funktionalität dynamischer Klassen und dynamischer Instanzen ab. Zum Teil werden ähnliche Dinge benötigt wie bei redefinierbaren Klassen und deren Instanzen. Daher wurden in MCS 1.0 beide Erweiterungen zu einer zusammengefaßt, siehe Seite 208. Das heißt, daß Instanzen redefinierbarer Klassen dynamisch reklassifizierbar sind. Die Funktion `change-class` bewirkt den gewünschten Effekt, falls es sich um eine Instanz einer redefinierbaren Klasse handelt und meldet sonst einen Fehler.

[10] Hierbei werden nicht nur Instanzen dynamisch klassifiziert, sondern auch Subsumtionsbeziehungen zwischen Klassen festgestellt bzw. abgeleitet.

8.3.4 Methodenkombination

Das Konzept der Methodenkombination wurde bereits in Kapitel 3.4.3 auf Seite 57, in Kapitel 4.2.2 auf Seite 81 sowie in Kapitel 6.2.2 auf Seite 157 angesprochen. Die Vereinfachung des Konstrukts `call-next-method` in MCS 1.0 hat bereits zur signifikanten Effizienzsteigerung der Methodenkombination geführt, siehe Seite 214. Bevor ich nun ein neues Konzept der Methodenkombination vorstelle, fasse ich die bisherigen Konzepte zusammen.

Ausgangskonzepte

Das älteste und bewährteste Konzept stammt aus Smalltalk. Abbildung 8.32 zeigt den Kontrollfluß[11] von den spezielleren Methodenaufrufen zu den allgemeineren sowie ihre Ergebnisablieferung. Die Steuerung der Aufrufe liegt bei spezielleren Methoden.

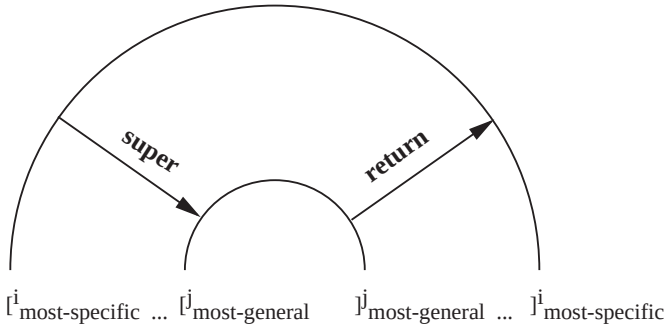


Abbildung 8.32: Modell der einfachen Methodenkombination mit dem Primitiv `super` aus Smalltalk.

Dieser Mechanismus wird in Java, CLOS, MCS 0.5, MCS 1.0 und auch im Kern von Τελος unterstützt. Eine umgekehrte Verantwortlichkeit wird mit dem Konstrukt `inner` aus BETA erreicht. Dabei entscheidet die generellere Methode, ob und wann eine speziellere Methode aufgerufen wird, siehe Abbildung 8.33.

Dieser Mechanismus steht dem Benutzer nur in BETA zur Verfügung. In CLOS und in Τελος kann er als Erweiterung realisiert werden. Das bisher ausdrucksstärkste Konzept der Methodenkombination bietet CLOS. Seine Standardmethodenkombination ist aber, wie man in der Abbildung 8.34 sieht, viel komplizierter als in BETA oder Τελος.

Es werden vier Arten von Methoden miteinander kombiniert. Der Steuerungsfluß wird zum Teil explizit beeinflußt (dies sind die gestrichelten Pfeile) und zum Teil implizit (angedeutet mit vollen Pfeillinien). Die Pfeilbeschriftung ε markiert automatische Übergänge von einer Methode zur nächsten. In beiden Varianten der einfachen Methodenkombination (`super` und `inner`) erfolgt der Aufruf der nächsten Methode explizit im Programmtext. In CLOS sind die Aufrufe von Before- und After-Methoden implizit. Falls es aber Around-Methoden gibt, ist der Übergang von der generellsten Around-Methode zur speziellsten

[11] Die treffendere Übersetzung müßte eigentliche Steuerungsfluß lauten.

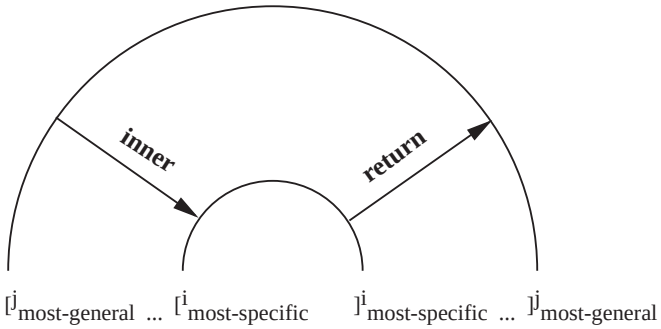


Abbildung 8.33: Modell der einfachen Methodenkombination mit dem Primitiv `inner` aus BETA.

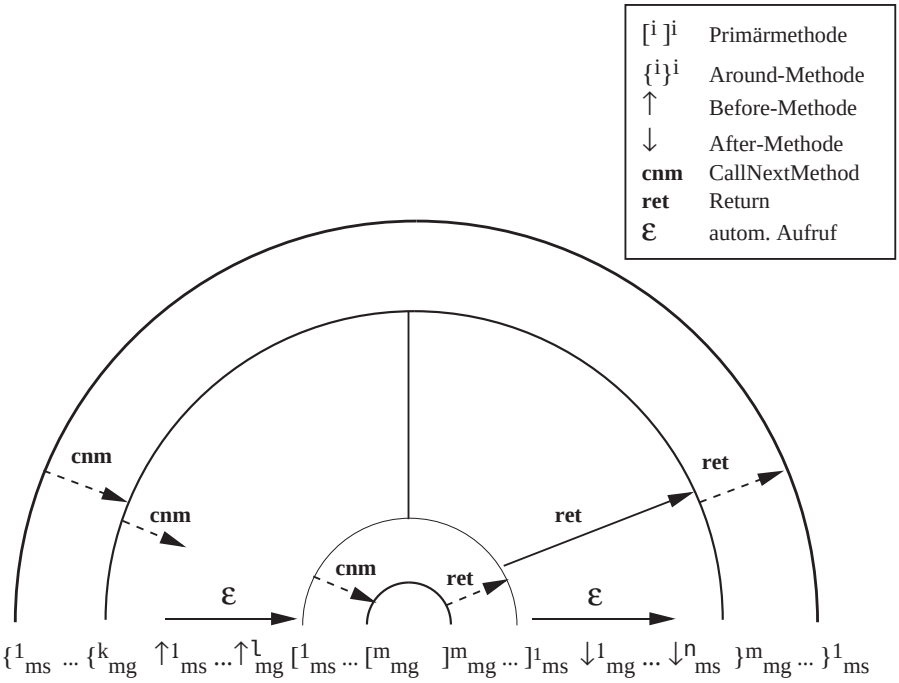


Abbildung 8.34: Modell der Standardmethodenkombination aus CLOS.

Before-Methode explizit. Es ist also kein Wunder, daß die Compilerbauer teilweise zehn Jahre gebraucht haben, um diesen Mechanismus effizient zu implementieren, und andere es immer noch nicht geschafft haben, siehe Performanzmessungen auf Seite 258. Ebenso zeigt die praktische Erfahrung, daß die unbedachte Verwendung sämtlicher Möglichkeiten der Standardmethodenkombination in CLOS die Verständlichkeit und Robustheit von Programmen erschwert.

Relevante Designfragen

Will man die Grundidee von abstrakteren Mechanismen des Spezialisierens und Generalisierens von Objektverhalten nicht gänzlich mit der mißlungenen CLOS-Lösung über Bord werfen, so müssen zunächst die in diesem Zusammenhang relevanten Fragen gestellt und beantwortet werden. Aus Programmierersicht stellen sich folgende konzeptionelle Fragen:

- Kann für eine Klasse nur genau eine Methode zum Tragen kommen oder können es mehrere sein? Im Sinne der Wiederverwendbarkeit und besseren Wartbarkeit sollte letzteres der Fall sein. Dies ist die grundlegende Frage, ob es Methodenkombination geben soll.
- Darf es, oder gar muß es, eine speziellere Methode in der Subklasse geben? Je nach Aufgabe können beide Fragen mit ja oder mit nein beantwortet werden, d. h. alle drei Varianten können sinnvoll sein.
- Kann die Methodenkombination *explizit* oder auch *implizit* erfolgen? Je nach Aufgabenstellung ist das eine oder das andere angemessen. Man braucht daher beide Möglichkeiten. Die explizite Kombination sollte aber dennoch abstrakt, also z. B. mit `callNextMethod`, erfolgen und nicht wie in C++, wo die genaue Klasse einer Methode angegeben werden muß.
- Erfolgt die Regie der Kombination aus Sicht des *Providers* wiederverwendbarer Komponenten (`inner` in BETA) oder aus Sicht des *Nutzers*, der vorhandene Komponenten wiederverwendet (`super` in Smalltalk, JAVA und TELOS-Kern)? Auch hier braucht man je nach Aufgabe beide Möglichkeiten.
- Handelt es sich bei einigen Methoden um *Vor-* oder *Nachberechnungen* (Before- bzw. After-Methoden in CLOS), bezogen auf die Aktionen der Super- bzw. Subklasse? Dies sollte man sowohl explizit als auch deklarativ ausdrücken können.
- Sollen die spezielleren Methoden zuerst ausgeführt werden oder umgekehrt? Auch hier kann beides sinnvoll sein.
- Müssen die Methoden ihre *Berechnungsergebnisse* miteinander kommunizieren? Und wenn ja, wie können sie das tun? Bei expliziter Kombination liefert `call-next-method` das Ergebnis der nächstallgemeineren bzw. nächstspezielleren Methode. Bei impliziter Kombination sollten die Ergebnisse unabhängig von einander sein.

Erweiterung um das Inner-Konstrukt aus BETA

Vom ΤΕΛΟΣ-Kern wurde zunächst nur die explizite Methodenkombination mit vorgegebener Sortierung der Methoden nach ihrer Spezifizität (die speziellste zuerst) unterstützt. Als erste Erweiterung bietet es sich nun an, diese Reihenfolge über eine weitere Option generischer Funktionen festzulegen und somit auch die umgekehrte Reihenfolge zu ermöglichen, zum Beispiel:

```
(defgeneric f ((x <object>))
   method-order: less-specific-first)

(defgeneric g ((x <object>))
   method-order: more-specific-first)
```

Die Option `method-order:` hat den Default-Wert `most-specific-first`, was der bisherigen Interpretation von `call-next-method` entspricht, nämlich die nächstallgemeinere Methode aufzurufen. Gibt man jedoch als Wert `most-specific-last` an, so erhält man in Verbindung mit `call-next-method` das `inner`-Konstrukt aus BETA, weil in einer allgemeineren Methode die nächstspeziellere Methode aufgerufen wird. Diese Erweiterung fügt sich so harmonisch in das bisherige Konzept ein, daß sie in den Objektsystemkern aufgenommen werden kann. Auch die Implementierung braucht dafür nur geringfügig erweitert zu werden. Es reicht aus, die anwendbaren Methoden in umgekehrter Reihenfolge zu sortieren.

Die Frage nach der Kommunizierbarkeit der Ergebnisse zwischen Methoden ist bei expliziter Methodenkombination leicht beantwortet: `call-next-method` liefert das Ergebnis der nächstallgemeineren bzw. nächstspezielleren Methode.

Ob es sich bei einer `call-next-method` aufrufenden Methode um Vor- oder Nachberechnungen (oder beides) handelt wird prozedural durch die Aufrufstelle im Methodenrumpf zum Ausdruck gebracht. Diese Eigenschaft ist vom System im allgemeinen nicht ableitbar, wohl aber in den meisten praktisch relevanten Fällen. Die implizite Kombinationsart mit Before- und After-Methoden kann beispielsweise wie folgt ausgedrückt werden:

```
(defmethod f ((x A))      ; simulates a CLOS :before method
  ;; pre-computations
  ...
  (call-next-method))

(defmethod g ((x A))      ; simulates a CLOS :after method
  (prog1
    (call-next-method)
    ;; post-computations
    ...))

(defmethod gf ((x A))      ; simulates a CLOS :around method
  ;; pre-computations
  ...
```

```
(prog1
  (call-next-method)
  ;; post-computations
  ...))
```

Diese Simulation der CLOS-Funktionalität bei angenommenem Wert `most-specific-first` für die Option `method-order:` war natürlich auch ohne die eben diskutierte Erweiterung möglich. Nimmt sie hingegen den Wert `most-specific-last` an, so ergibt sich eine neue Funktionalität, die über CLOS hinausgeht. Das Konstrukt `prog1` ist hier notwendig, um dafür zu sorgen, daß das richtige Ergebnis zurückgeliefert wird. Mit dieser Simulation bleibt aber der Einwand bestehen, sie sei nicht deklarativ im Vergleich zu CLOS. Daher stelle ich im nächsten Unterabschnitt die Erweiterung um implizite Methodenkombination vor.

Erweiterung um implizite Methodenkombination

Ein Beispiel für die implizite Methodenkombination sind die Before- und After-Methoden aus CLOS. Dort sind sie aber mit der expliziten Methodenkombination (Around- und Primärmethoden) verquickt, was die Semantik und Implementierung kompliziert. Nimmt man die Kriterien der Orthogonalität und Einfachheit ernst, so ist es besser beide Konzepte zu trennen. Eine generische Funktion sollte entweder dem einen oder dem anderen Prinzip folgen, nicht aber beiden gleichzeitig.

Die Grundidee der *impliziten Methodenkombination* liegt darin, daß alle anwendbaren Methoden zur Ausführung kommen. Die Frage der Reihenfolge ist hierzu orthogonal, d. h. sowohl *die speziellste zuerst* als auch *die allgemeinste zuerst* können sinnvoll sein. Schwieriger ist hier die Frage nach der Kommunizierbarkeit der Methodenergebnisse. Am einfachsten ist der Fall, wenn die Berechnungen einer Methode unabhängig von Ergebnissen vorher ausgeführter Methoden sind. Dann stellt sich nur die Frage nach dem Gesamtergebnis eines Aufrufs der generischen Funktion.

CLOS stellt hierfür die sogenannten *einfachen Kombinationsarten* zur Verfügung (*simple built-in method combination types*). Diese sind nach den Operationen benannt, die zum Schluß auf allen Ergebnissen ausgeführt werden und das Gesamtergebnis bilden: `+`, `and`, `append`, `list`, `max`, `min`, `nconc`, `or` und `progn`. Negativ zu bewerten ist hier die Möglichkeit, die Grundidee außer Kraft zu setzen, indem auch bei diesen Kombinationsarten Around-Methoden definiert werden dürfen, so daß die sonst implizit aufzurufenden Primärmethoden explizit aufgerufen werden müssen, um überhaupt noch zur Anwendung zu kommen. Eine weitere Entscheidung in CLOS, die Option `:most-specific-first` bzw. `:most-specific-last` nur auf die Primärmethoden und nicht auf die Around-Methoden[12] anzuwenden, widerspricht klar meinen Designkriterien.

Die Erweiterung um die implizite Methodenkombination soll den Kriterien der Orthogonalität, der Uniformität, Einfachheit, Robustheit und Effizienz gerecht werden. Daher werden hier zwei weitere Optionen von `defgeneric` eingeführt:

[12] Speziellere Around-Methoden werden bei den einfachen Kombinationsarten wie bei der Standardkombination auf jeden Fall zuerst ausgeführt.

- `method-combination:` mit den möglichen Werten `implicit` und `explicit`, letzterer ist der Default-Wert, und
- `operator:` mit den möglichen Werten `+`, `and`, `append`, `list`, `max`, `min`, `nconc`, `or`, `progn`und `prog1`[13]. Letzterer ist der Default-Wert und liefert das Ergebnis der zuerst ausgeführten Methode zurück.

In Verbindung mit der ersten Erweiterung um die Option `method-order:` ergibt sich z. B. folgende Möglichkeit, die Initialisierung von Objekten nach dem JAVA-Konzept anzubieten. Dabei werden die Initialisierungsmethoden implizit ausgeführt, beginnend mit der allgemeinsten:

```
(defgeneric initialize ((x <object>) initargs)
   method-combination: implicit
   method-order: most-specific-last)
```

Entsprechend den sogenannten JAVA-Konstruktoren sorgt hier die implizite Methodenkombination dafür, daß alle Methoden ausgeführt werden, und zwar in der Reihenfolge *die speziellste zuletzt*. Das Ergebnis der allgemeinsten Methode bildet das Gesamtergebnis eines Aufrufs von `initialize`.

Das im ΤΕΛΟΣ-Kern spezifizierte Verhalten der Initialisierung entspricht folgender Definition:

```
(defgeneric initialize ((x <object>) initargs)
   method-combination: explicit
   method-order: most-specific-first)
```

Weitere Kandidaten für die implizite Methodenkombination sind solche generische Funktionen, die vom System vorgegeben sind und interne Operationen aufrufen, die dem Benutzer nicht offengelegt werden sollen, z. B. die internen Schreiboperationen auf Metaobjekten. In ΤΕΛΟΣ ist neben den Initialisierungsmethoden die Operation `computeAndEnsureFieldAccessors` so ein Kandidat. Aber auch in CLOS MOP gibt es Operationen, für die in der Dokumentation die Auflage spezifiziert wird, daß in spezielleren, benutzerdefinierten Methoden auf jeden Fall `call-next-method` aufzurufen sei und daß dessen Ergebnis als Gesamtergebnis zurückzuliefern sei. Hier stellt sich die Frage, warum die CLOS-Designer nicht das Mittel der Methodenkombination eingesetzt haben, um solche Auflagen vom System sicherzustellen. Klar ist, daß die Standardmethodenkombination von CLOS dies nicht leisten kann.

Die vorgeschlagenen Erweiterungen der Methodenkombination in ΤΕΛΟΣ zeigen, wie die Ausdruckskraft der vordefinierten Kombinationsarten in CLOS erreicht und übertroffen werden kann, ohne dessen Nachteile in Kauf nehmen zu müssen[14].

[13] Auch benutzerdefinierte Operatoren können hier zugelassen werden, worauf ich aber nicht näher eingehe.

[14] Natürlich kann man in CLOS mit dem Konstrukt `define-method-combination` die hier beschriebenen Mechanismen auch bereitstellen. Meine Kritik an CLOS richtet sich gegen das als Standardfall spezifizierte Verhalten.

8.3.5 Wissensrepräsentation mit Konzepten und Relationen: TINA

Weitergehende Konzepte wie sie aus Wissensrepräsentationssprachen bekannt sind, wurden in ΤΕΛΟΣ harmonisch eingeordnet und als Erweiterung der Programmiersprache realisiert [Kopp, 1996a]. Dies bietet viele Vorteile, die ich in Kapitel 2 schon angesprochen habe. Beispielsweise können die Werkzeuge einer Programmentwicklungsumgebung sowie Bibliotheken der Programmiersprache auch als Hilfsmittel zur Wissensrepräsentation bzw. zur Konstruktion wissensbasierter Systeme benutzt werden. Aber am besten zitiere ich hier, was Jürgen Kopp als Resultat seiner Dissertation sehr prägnant formuliert hat [Kopp, 1996a, S. 5]:

> - "Die Sprachmittel von TINA unterstützen die abstrakte, ausdrucksstarke Modellierung einer Domäne. Modellierungen können beginnend mit der Analyse und ersten Spezifikation des Gegenstandbereiches sukzessive verfeinert und erweitert werden. Die Sprachmittel der Implementierungssprache werden für TINA-anwendungen durchgereicht und wiederverwendet. TINA-Anwendungen sind ebenfalls wiederverwendbar.
> - Die spezielle Implementierungstechnik der Sprache TINA macht ihre Konzeption und Implementierung sehr kompakt. TINA wird mit Hilfe eines Spracherweiterungsprotokolls in eine existierende Programmiersprache [ΤΕΛΟΣ Anm. des Autors] eingebettet. Dabei wird die Implementierung dieser Programmiersprache wiederverwendet und erweitert.
> - Die benutzte Implementierungstechnik erlaubt das Wiederverwenden der TINA-Implementierung, was durch die Erweiterung der Ausdrucksmöglichkeiten von TINA dokumentiert wird.
> - Modellierungen in der Wissensrepräsentationssprache TINA sind ohne zusätzliche Mittel in schlüsselfertige Applikationen überführbar. Dadurch werden die Applikationen portabel und ihr Ressourcenbedarf wird geringer, so daß sie auch auf PCs ablauffähig sind."

Aus der Sicht des Entwurfs von ΤΕΛΟΣ und seiner Implementierung CELOS war es sehr hilfreich, die Konzepte und ihre Realisierung der Validierung durch eine reale Anwendung zu unterziehen. Schwachstellen konnten früh erkannt und beseitigt werden.

8.4 Vergleich von EuLisp mit Java

Der Überblick über JAVA wurde bereits in Kapitel 3.3.3 gegeben. In diesem Abschnitt vergleiche ich EULISP mit JAVA. Natürlich steht dabei der objektorientierte Sprachteil im Vordergrund. Aber auch andere Aspekte, insbesondere das Modulkonzept, müssen in den Vergleich einbezogen werden, weil Klassen in Java auch der Modularisierung dienen. Mit diesem Vergleich möchte ich klar machen, welche Designentscheidungen in JAVA aus meiner Sicht anders zu treffen wären. Da inzwischen kaum an gravierende Änderungen der Sprachdefinition zu denken ist, werden hier Hinweise darauf gegeben, wie man JAVA-Konstrukte verwenden sollte, um zumindest einige Vorteile objektorientierter Konzepte aus ΤΕΛΟΣ über entsprechende Programmierrichtlinien dennoch nutzen zu können.

Generell kann man feststellen, daß in JAVA einige grundlegende Sprachkonzepte mit EuLISP im Prinzip übereinstimmen. Dazu gehört die automatische dynamische Speicherverwaltung, die sogenannte *Pointer-Semantik* von Objekten und die daraus folgende Abwesenheit expliziter *Pointer*, im Unterschied zu C++, Pascal oder anderen Sprachen. Wie in EuLISP, stellen in JAVA die Sprachmittel zur Klassifizierung gegenüber solchen zur Verteilung und Parallelisierung orthogonale Konzepte dar. Auch der *Exception-Mechanismus* von JAVA entspricht weitgehend dem *Condition-System* aus EuLISP.

Ein wesentlicher Unterschied zwischen JAVA und EuLISP, ist die strenge Typisierung von Konstanten- und Variablenbindungen sowie von Funktionsergebnissen. Dadurch kann in JAVA zur Übersetzungszeit eine weitergehende Prüfung der Typkorrektheit von Programmen erfolgen.

Ein weiterer gravierender Unterschied liegt darin, daß JAVA keine syntaktische Spracherweiterung zuläßt. In EuLISP hingegen kann die Syntax der Sprache auf uniforme Weise durch Makrodefinitionen erweitert werden.

In den folgenden Abschnitten werden die objektorientierten Konzepte beider Sprachen genauer verglichen.

8.4.1 Klassen vs. Module

In Kapitel 4.1.4 habe ich die Gründe diskutiert, warum es wichtig ist, zwischen der Modularisierung von Programmen und der Klassifizierung von Objekten zu unterscheiden. Diese Unterscheidung führt in EuLISP zu zwei orthogonalen, d. h. sich ergänzenden, Konzepten. In Java hingegen gibt es nur Klassen. Sie dienen sowohl der Klassifizierung von Objekten als auch der Modularisierung von Programmen. Der Vermeidung von Namenskonflikten dienen zusätzlich sogenannte *Packages*. Jede Klasse wird einem Package zugeordnet. Sichtbarkeitsdirektiven in Klassendefinitionen wie `public`, `private`, etc. beziehen sich auf das Package, dem die Klasse zugeordnet ist. Ein Package bündelt implizit die Exports aller zu ihm gehörenden Klassen und bildet die Sichtbarkeitsgrenze derjenigen Bestandteile, die nicht als `public`) deklariert wurden. Import-Anweisungen sind aber nicht auf Packages bezogen, sondern auf eine Quellcode-Datei, in der sich Klassendefinitionen befinden. Import-Anweisungen dienen auch nur der kürzeren Schreibweise von Namen. Sie haben keine semantische Bedeutung, da Klassen, Methoden und Felder über ihre vollständige Namen, d. h. mit dem Packagenamen als Präfix, auch ohne entsprechende Import-Anweisungen referiert werden dürfen. Hier ist das Modulkonzept von EuLISP strikter. Es verlangt explizite Import-Anweisungen.

Obwohl es in JAVA kein explizites Konstrukt für Module gibt, kann man doch typische Klassen identifizieren, die reinen Modulcharakter haben. Dazu gehören alle Klassen, die nur *Klassenvariablen* bzw. *Klassenmethoden* (`static`) besitzen und die nie instanziiert werden, z. B. `java.lang.Math`. Da es gute Gründe gibt, zwischen Klassen- und Modulbildung zu unterscheiden, selbst wenn es in einer Sprache nur ein syntaktisches Konstrukt für beides gibt, sollte im Anwendungsdesign dokumentiert werden, welches die Module und welches die Datentypen sind. Als Programmierrichtlinie sollten Module als JAVA-Klassen ohne Instanzvariablen und ohne Konstruktoren implementiert werden. Diese besitzen dann nur Klassenvariablen und -Methoden, also solche, die als `static` deklariert werden. Die umgekehrte Forderung, Datentypen ohne Klassenvariablen und ohne Klassenmethoden zu

implementieren, könnte sich als zu puristisch erweisen und den Nachteil mit sich bringen, daß zu viele Klassen definiert werden müßten. Dies könnte aber eine "weiche" Programmierrichtlinie sein, von der mit Begründung abgewichen werden darf. Natürlich ist dabei zu beachten, daß Klassenmethoden die wichtigste Eigenschaft der Objektorientierung fehlt, nämlich ihre Spezialisierbarbeit. Sie entsprechen somit nicht-generischen Funktionen in TELOS, siehe unten.

Ein Beispiel dafür, wie fragwürdig die Vermischung des Modul- und des Objektsystems ist, zeigt die Behandlung von Klassen durch die automatische Speicherverwaltung. Klassen unterliegen per Standardeinstellung des Laufzeitsystems der automatischen Speicherverwaltung. Dies führt dazu, daß beim Zugriff auf statische Variablen einer Klasse, diese plötzlich neu initialisiert werden, weil es die Klasse nicht mehr gab und sie neu geladen wurde. Wurden statische Variablen vom Programm zwischendurch gesetzt, so kann der entsprechende Wert einfach verschwunden sein. Die Frage stellt sich hier, unter welchen Voraussetzungen eine Klasse dem freien Speicher zugeführt werden darf. Das zählen von Referenzen auf das Klassenobjekt ist in meinen Augen nicht ausreichend. Ob eine Klasse nicht mehr benötigt wird, hängt nicht nur von Objektreferenzen ab, sondern auch vom potentiellen dynamischen Programmablauf. Falls es einen Berechnungspfad gibt, bei dem ein Zugriff auf die statischen Variablen einer Klasse erfolgt, kann sie nicht als "Müll" gelten. Selbst wenn diese Analyse entscheidbar wäre, kann sie von einem *Garbage Collector* kaum effizient genug zur Programmausführungszeit durchgeführt werden.

Betrachtet man eine Klasse als Modul bzw. als Programm, so gleicht das Standardverhalten von JAVA einem willkürlichen Beenden von Programmteilen und ihrem erneuten starten zwischendurch, ohne daß entsprechende Zwischenzustände gerettet werden. Dies haben JAVA-Designer wohl nicht mit Robustheit von JAVA-Anwendungen gemeint. Als ein *Work-Around* kann man allen JAVA-Programmierern nur empfehlen, die Option `-noclassgc` im Aufruf des JAVA-Laufzeitsystems, z. B. mit `java`, explizit zu setzen. Allerdings werden dadurch alle Klassen der automatischen Speicherverwaltung entzogen, auch solche, die nur in der Rolle als Datentyp verwendet werden.

8.4.2 Vergleich der Objektebene

In JAVA wird nicht explizit zwischen der Objekt- und der Metaobjektebene der Sprache unterschieden. Aber JAVA ist ähnlich wie EULISP strukturiert. Zur Strukturierung werden hier Packages verwendet, während es in EULISP die Module sind. Der Vorteil von EULISP ist, daß auch syntaktische Konstrukte in die Strukturierung einbezogen werden können. In JAVA sind es nur die Klassen, ihre Felder und Methoden. In JAVA kann das Package `java.lang.reflect` sowie die Klasse `java.lang.Class` der Metaobjektebene zugeordnet werden. Streng genommen muß auch die Methode `getClass()` der Klasse `Object` der Metaobjektebene zugerechnet werden. Dies kann jedoch mit JAVA-Mitteln nicht ausgedrückt werden, da Klassen nur als ganzes ex- bzw. importiert werden können. Alle anderen Packages ordne ich der Objektebene zu.

Vererbung

JAVA unterstützt wie der TELOS-Kern einfache Vererbung. Eine Erweiterung der Sprache um Mixin- oder allgemeine multiple Vererbung ist mit portablen Mitteln nicht möglich.

Die Tatsache, daß *Interfaces* als spezielle Typen bzw. Klassen mehrere Super-Interfaces haben können, hilft auch nicht weiter. Diese Einschätzung wird auch in [Flatt *et al.*, 1998] vertreten, wo ein spezielles Konzept der Mixins als Änderung von Java vorgeschlagen wird. Dabei stehen die Aspekte des Spezialisierens und Generalisierens von Objektklassen sowie ihre effiziente Implementierung nicht im Vordergrund.

Spezialisierung von Feldannotationen

Ein Feld zu spezialisieren bedeutet vor allem, daß sein Wert in Instanzen einer Subklassen von einem spezielleren Typ sein soll und daß der Ersatzwert entsprechend spezieller sein muß als in der Superklasse. Dies betrifft also die Feldannotationen `type:`, `initValue:`, `initFunction:` und `initExpression:`. Dabei sollen die generellen Methoden der Superklasse mit dem spezielleren Feld problemlos umgehen können. In JAVA ist dies leider nicht der Fall. Der Versuch, Feldannotationen zu spezialisieren, führt in JAVA zu einem zweiten Feld mit dem gleichen *einfachen* Namen[15]. Für speziellere Methoden ist nur das zweite Feld sichtbar, während für die Methoden der Superklasse nur das erste Feld sichtbar ist. Diese Strategie wird in Java *Shadowing* genannt. Das *Overriding*, als ein Mechanismus, die speziellsten Eigenschaften eines Objekts zur Laufzeit wirken zu lassen, ist aber eine notwendige Voraussetzung, um Struktur und Verhalten von Objekten adäquat und möglichst deklarativ zu spezialisieren. JAVA entspricht daher in diesem Punkt weder meinen Designkriterien aus Kapitel 2 noch den Konzepten aus Kapitel 4.

Die Beispiele in Abbildung 8.35 vermitteln eine Vorstellung, wie Vererbung mit Overriding und Shadowing in JAVA funktioniert.

Objekterzeugung und Initialisierung

Auf der *gedachten* Objektebene von JAVA werden neue Objekte über das syntaktische Konstrukt `new`, gefolgt vom sogenannten *Konstruktor-Namen* und geforderten Argumente, erzeugt. Aus der Sicht von TELOS sind JAVA-Konstruktoren als Initialisierungsmethoden anzusehen. Sie werden syntaktisch etwas anders definiert als alle anderen Methoden: Es wird für sie kein Ergebnistyp spezifiziert. JAVA-Konstruktoren der Superklassen werden automatisch zuerst ausgeführt. Das heißt, daß jeder Konstruktor implizit (oder auch explizit) als erste Anweisung ein `super()` enthält. Es werden also zuerst die generellen Initialisierungsaktionen durchgeführt und dann die spezielleren. In TELOS liegt es im Ermessen des Programmierers zu entscheiden, ob und wann die allgemeineren `initialize` Methoden über `call-next-method` zum Zuge kommen sollen oder nicht. In Java kann man nicht verhindern, daß alle anwendbaren Initialisierungsmethoden, dies sind alle Konstruktoren mit der gleichen Parameterliste, automatisch ausgeführt werden.

Ein weiterer Unterschied zwischen TELOS und JAVA liegt darin, daß es in Java keine optionalen Parameter gibt. Dort müssen daher meistens mehrere Konstruktoren definiert werden. Die Java-Lösung zieht es nach sich, daß das Initialisierungsverhalten, selbst innerhalb einer Klasse, auf mehrere Konstruktoren aufgeteilt ist, was das Programmverstehen und die Programmwartung erschwert.

[15] Der *vollständige* Feldname enthält als Präfix den Klassennamen.

```
class A {
  String x = "A";
  String f() {return x;}
  String g() {return x;}
  }
class B extends A {
  String x = "B";
  String g() {return x;}
  }
A a = new A();
B b = new B();
A c = (A) b;

a.x;         // ==> A
b.x;         // ==> B    shadowing
c.getClass;  // ==> B
c.x;         // ==> A    shadowing and casting, NON-INTUITIV
c.f();       // ==> A    inherited method from A gets x of A, NON-INTUITIV
c.g();       // ==> B    overriding method of B gets x of B
```

Abbildung 8.35: Shadowing und Overriding in Java

Generische Operationen und Methoden

Im Unterschied zu ΤΕΛΟΣ gibt es in JAVA keine generischen Operationen, sondern nur Methoden. Statt dem generischen Dispatch spricht man von *Methoden-Overriding*. Dies entspricht dem Konzept des *Message-Passing* in MCS 0.5, nur daß die Notation ohne das syntaktische Konstrukt `send` erfolgt:

`object.`*method-name*`(`*argument** `)`

Wie schon bei JAVA-Konstruktoren kann es in einer Klasse mehrere Methoden mit dem gleichen Namen, aber unterschiedlichen Parameterlisten geben. Solche Methoden haben nichts mit einander zu tun. Problematisch ist, daß auch Parameterlisten mit der gleichen Anzahl von Parametern, aber spezielleren oder generelleren Typen (bzw. Klassen) als unterschiedlich betrachtet werden. Dies erschwert die Spezialisierung von Methoden. Oft erhält das *Empfänger-Objekt* auch speziellere Argumente. Dies darf jedoch nicht im Methodenkopf spezifiziert werden. Stattdessen muß die Parameterliste der Superklasse beibehalten werden und im Methodenrumpf muß für die bekanntermaßen spezielleren Argumente dann ein sogenanntes *casting* erfolgen.

Dieses Verhalten hängt natürlich damit zusammen, daß es in JAVA keine Multimethoden gibt, was seinerseits aus der Beschränkung herrührt, daß eine Methodendefinition sich innerhalb genau einer Klasse befinden muß. Dies ist eine Spätfolge der Doppelrolle von Klassen als Typ und als Modul.

Aber auch bzgl. des Empfängerobjekts wird nicht immer die speziellste anwendbare Methoden ausgewählt, sondern eine generellere, falls die Typdeklarationen von Variablen eine Vorentscheidung zur Übersetzungszeit nicht zulassen. Dies ist ein sehr unintuitives Dispatch-Verhalten und führt zu schwer auffindbaren Programmfehlern.

Wie ΤΕΛΟΣ bietet JAVA einen Mechanismus, in einer spezielleren Methode die nächst allgemeinere Methode auszuführen, und zwar mit dem syntaktischen Konstrukt `super()`, das dem `call-next-method` in ΤΕΛΟΣ entspricht.

8.4.3 Vergleich der Metaobjektebene

Schon in der ersten Sprachdefinition von JAVA gab es einige reflektive Operationen. Dazu zählt die Methode `getClass()` der Klasse `java.lang.Object`. Weitere typische Sprachmittel der Metaobjektebene wie

- `newInstance()`,
- `getName()`,
- `getSuperclass()`,
- `getInterfaces()`,
- `isInterface()`,
- `forName(String className)`,
- ...

stellt die Klasse `java.lang.Class` zur Verfügung. Mit JDK1.1 wurde im Package *java.lang.reflect* ein umfassendes *Introspektionsprotokoll* spezifiziert. Dieses erlaubt, ein beliebiges Objekt, seine Klasse und somit alle wesentlichen Elemente, die die Struktur und das Verhalten des Objekts bestimmen, zu inspizieren. Hierfür wurden zusätzlich zu `Class` weitere Metaobjektklassen wie `Field`, `Constructor`, `Method` definiert. Generische Werkzeuge wie Klassenbrowser, Inspektoren, Debugger etc. können somit in JAVA portabel implementiert werden.

Im Unterschied zu TELOS sind aber alle Metaobjektklassen in JAVA *final* und können somit nicht spezialisiert werde. Neuartige Klassen, z. B. mit multipler Vererbung, Multimethoden oder spezielle Feldannotationen können nicht als Spracherweiterung realisiert werden. Die Sprache JAVA kann im objektorientierten Teil nicht selbtst wiederverwendet bzw. spezialisiert werden. Sie erfüllt nicht meine Designkriterien aus Kapitel 2.

8.4.4 Abschlußeigenschaften

JAVA unterstützt das explizite Abschließen von Programmeinheiten [Gosling *et al.*, 1996]. So können Klassen als *final* deklariert werden, was ihre Spezialisierung unterbindet. Damit sind automatisch auch alle Methoden einer abgeschlossenen Klasse abgeschlossen, da es keine Methoden außerhalb der syntaktischen Klammer einer Klassendefinition geben darf. Zusammen mit der strengen Typisierung sind daher gute Voraussetzungen für Optimierungen zur Übersetzungszeit geschaffen worden. Gegenwärtige Compiler und JAVA-Laufzeitsysteme erfüllen aber noch nicht die Performanz-Erwartungen der Programmierer und Anwender.

TELOS ist zwar nicht streng typisiert, unterstützt jedoch besser als CLOS den Einsatz traditioneller Compilerbautechnik, wie Typinferenz, so daß hier ähnliche Voraussetzungen wie in JAVA geschaffen wurden und auf jeden Fall Vorteile gegenüber CLOS zu erwarten sind.

8.5 Bewertung von TELOS

Im Unterschied zu MCS 0.5 und MCS 1.0 konnte das Sprachdesign von TELOS ohne Rücksicht auf die Kompatibilität mit anderen Objektsystemen vorgenommen werden. Ein gewisser Rahmen wurde natürlich mit der Basissprache EULISP vorgegeben, deren integraler Bestandteil TELOS wurde. Im Unterschied zu COMMONLISP und CLOS wurden EULISP und TELOS gleichzeitig entworfen. Die Darstellung von TELOS in diesem Kapitel baut auf den Konzepten aus Kapitel 4 und den Implementierungstechniken aus Kapitel 5 auf.

Vergleicht man TELOS mit den beiden MCS-Versionen, so werden drei Tendenzen deutlich:

Die *erste Tendenz* betrifft den *Umfang der Funktionalität der Objektsysteme*. Hier ergibt sich eine *Bogenbewegung*: MCS 0.5 beschränkte sich auf die Kernfunktionalität wesentlicher Konzepte wie

- vereinfachte multiple Vererbung aus Kompatibilitätsgründen mit FLAVORS,
- Message-Passing mit einfachem Dispatch,

- Methodenkombination, orientiert an der Standardmethodenkombination von CLOS,
- Reflektion, bezogen auf Klassen (daher auch der Name *Meta Class System*), mit einfachen und implementierungsabhängigen Erweiterungsmöglichkeiten.

In MCS 1.0 wurde die Funktionalität insgesamt erweitert um:

- Mixin-Vererbung bereits im Kern,
- generische Funktionen mit mehrfachem Dispatch,
- generische Lese- und Schreiboperationen,
- Reflektion, bezogen auf Klassen und Feldannotationen, mit an CLOS MOP orientierten Erweiterungsmöglichkeiten.
- Redefinieren von Klassen mit automatischer Anpassung abhängiger Objekte,
- Reklassifizieren von Objekten.

Im ΤΕΛΟΣ-Kern wurde die unmittelbar angebotene Funktionalität reduziert auf:

- einfache Vererbung,
- nicht-generische Lese- und Schreiboperationen,
- einfache Methodenkombination,
- keine Reflektionsmittel auf der Objektebene, sondern nur auf der Metaobjektebene, und zwar bezogen auf Klassen, Feldannotationen, generische Funktionen und Methoden mit ausgeprägten Erweiterungsmöglichkeiten.

Die *zweite Tendenz* betrifft die *Komplexität der einzelnen Sprachkonstrukte.* Dies ist eine abfallende Linie. Oder wenn man es positiv ausdrücken will: die Sprachkonstrukte wurden immer einfacher. Aus multipler Vererbung in MCS 0.5 wurde über die Mixin-Vererbung in MCS 1.0 schließlich nur noch einfache Vererbung im Kern von ΤΕΛΟΣ. Die Methodenkombination begann in MCS 0.5 mit Before-, After-, Around- und Primärmethoden sowie mit optionalen Argumenten für `call-next-method`; letztere wurden in MCS 1.0 aufgegeben. In ΤΕΛΟΣ wird im Kern nur noch einfache Methodenkombination angeboten, d. h. es gibt nur noch Primärmethoden und das Konstrukt `call-next-method` ohne optionale Argumente. Selbst die vorgeschlagenen Erweiterungen um die Optionen zur umgekehrten Reihenfolge der anwendbaren Methoden sowie um die implizite Methodenkombination mit Operatoren kommen mit *einer* Art von Methoden aus.

Die *dritte Tendenz* betrifft die *Erfüllung der Designkriterien.* Hier ist eine eindeutige Steigerung zu erkennen. Die stärkere Orthogonalität der Konstrukte und ihre Vereinfachung haben unmittelbar zur höheren Robustheit und Effizienz beigetragen. Die Separierung aufwendiger Funktionalität wie des Redefinierens in MCS 1.0 sowie die Beschränkung auf einfache Vererbung im Kern von ΤΕΛΟΣ entsprechen dem Verursacherprinzip. Kosten

entstehen nur da, wo sie gerechtfertigt sind. Die schrittweise Systematisierung der Reflektionsmittel und die Bereitstellung eines Metaobjektprokolls mit mehreren Granularitätsstufen verbessert die Erweiterbarkeit des Objektsystems und somit auch die Entwicklungseffizienz komplexer Anwendungen. Die Einführung der Mixin-Vererbung im MCS-Kern und als ΤΕΛΟΣ-Erweiterung trägt signifikant zur besseren Klassifikation sowie Spezialisierung und Generalisierung von Objektklassen bei – der wichtigsten Aufgabe von Objektsystemen. Nicht zu unterschätzen ist auch das Modulkonzept von EuLisp, das als orthogonales Sprachmittel dazu beiträgt, das Verursacherprinizip explizit auf die Ebene der Programmiersprache zu bringen. Darüberhinaus sind Module ein Schlüsselkonzept zur Unterstützung der Modul- und Applikationskompilation. ΤΕΛΟΣ nutzt das Modulkonzept zur eigenen Strukturierung und zur Entlastung von Aufgaben wie Sichtbarkeitskontrolle, Import/Export-Spezifikation, Sicherstellung von Abschlußeigenschaften etc.

Alle oben genannten Aspekte werden in folgender Tabelle 8.1 der Designkriterien im Vergleich mit Java und CLOS zusammengefaßt.

Kriterium	Java	CLOS	MCS 0.5	MCS 1.0	TELOS
Abstraktion	–	++	+	++	++
Klassifikation	+	++	+	++	++
Spezialisierung	– –	+	+	++	++
Komposition	–	+	+	+	++
Modularisierung	+	–	–	–	++
Orthogonalität	–	–	+	+	++
Einfachheit	+	– –	+	+	++
Reflektion	–	++	+	+	++
Erweiterbarkeit	– –	++	+	+	++
Robustheit	++	–	+	+	++
Effizienz	–	–	+	+	++
Verursacherprinzip	+	–	+	+	++

Tabelle 8.1: Vergleich der Sprachen anhand der Designkriterien.

Die Tabelle 8.2 zeigt die bereitgestellten Sprachkonzepte im Überblick.

Kürzel	Sprachkonzept	Java	CLOS	MCS 0.5	MCS 1.0	TELOS
Vererbung						
singlI	einfache Vererbung	ja	nein	nein	nein	im Kern
multI	multiple Vererbung	nein	ja	ja	ja	erweit.
mixI	Mixin-Vererbung	nein	nein	nein	ja	erweit.
Struktur						
instF	instanzspezifische Felder	ja	ja	ja	ja	ja
classF	klassenspezifische Felder	ja	ja	ja	ja	ja
sharedF	shared Slots	nein	ja	nein	nein	erweit.
extF	erweiterbare Felder	ja	ja	ja	ja	ja
specF	spezialisierbare Felder	nein	ja	ja	ja	ja
Verhalten						
MP	Message Passing	ja	erweit.	ja	erweit.	erweit.
GF	generische Funktionen	nein	ja	nein	ja	ja
singlD	einfacher Dispatch	ja	ja	ja	ja	ja
multD	mehrfacher Dispatch	nein	ja	nein	ja	ja
classD	klassenspez. Dispatch	ja	ja	ja	ja	ja
instD	instanzspez. Dispatch	nein	ja	nein	nein	erweit.
Methodenkombination						
explicitMC	explizit	ja	ja	ja	ja	ja
implicitMC	implizit	nein	ja	nein	nein	erweit.
superMC	super() aus Smalltalk	ja	ja	ja	ja	ja
demonMC	Standard aus CLOS	nein	ja	ja	ja	erweit.
innerMC	inner() aus BETA	nein	erweit.	nein	nein	erweit.
Dynamik						
dynC	Klassenwechsel	nein	ja	nein	erweit.	erweit.
redC	redefinierbare Klassen	nein	ja	zum Teil	erweit.	erweit.
Reflektion						
funR	funktionale Objekte	nein	ja	ja	ja	ja
introR	Introspektion	ja	ja	zum Teil	ja	ja
mopR	Metaobjektprotokoll	nein	ja	zum Teil	ja	ja
Syntax						
Syn	Syntax	C++	Lisp	Lisp	Lisp	Lisp
extSyn	erweiterbare Syntax	nein	ja	ja	ja	ja
Modularisierung						
Mod	Entität	Klasse Package	Datei Package	Datei	Datei	Modul
Hid	Sichtbarkeit	Klasse Package	Package	Package	Package	Modul
bindMod	Bindungsraum	Block	Block	Block	Block	Modul
nameMod	Namensraum	Package	Package	Package	Package	irrel.
Abschlußeigenschaften						
classBind	Klassenbindung	const	var	var	var	const
gfnBind	generische Funktionsb.	–	var	var	var	const
methBind	Methoden	const	var	var	const	const
sealC	Sealing von Klassen	ja	nein	nein	nein	zum Teil
sealGfn	Sealing von gen. Funkt.	–	nein	nein	nein	indirekt
sealGfn	Sealing von Methoden	ja	nein	nein	nein	indirekt

Tabelle 8.2: Vergleich der Sprachaspekte.

Kapitel 9

Zusammenfassung und Ausblick

Man muß nicht alles anders machen, aber vieles besser.

Autor unbekannt.

Ausgehend von den Defiziten existierender objektorientierter Programmiersprachen habe ich zunächst die Frage gestellt, *was effiziente Objektsysteme leisten sollen.* Um die richtige Antwort zu finden, bin ich von den Anforderungen komplexer, insbesondere wissensbasierter, Systeme ausgegangen und habe **12 Kriterien** formuliert, an denen alle Designentscheidungen zu messen sind:

- Abstraktion,
- Klassifikation,
- Spezialisierung,
- Komposition,
- Modularisierung,
- Orthogonalität,
- Einfachheit,
- Reflektion,
- Erweiterbarkeit,
- Robustheit,
- Effizienz und
- das Verursacherprinzip.

Diese Kriterien müssen auf unterschiedliche Weise berücksichtigt werden. Die Unterstützung der Klassifikation, der Spezialisierung, der Reflektion und der Erweiterbarkeit ist die ureigenste Aufgabe eines Objektsystems. Komposition und Modularisierung

stellen dazu orthogonale Prinzipien dar. Ein Modulkonzept sollte zusätzlich zum Objektsystem vorhanden sein und es sollte benutzt werden, um das Objektsystem zu modularisieren. Orthogonalität und Einfachheit beziehen sich auf die Gestaltung der Grundkonzepte und entsprechender Sprachkonstrukte. Ein Objektsystem muß robust gegenüber seinen Erweiterungen sein. Reflektions- und Spezialisierungsmittel dürfen die Integrität des Objektsystems und seiner Anwendungen nicht gefährden. Die Effizienz eines Objektsystems bezieht sich vor allem auf die statische und dynamische Objekterzeugung, auf Feldzugriffe und den generischen Dispatch einschließlich der Methodenkombination. Um ein effizientes Verhalten komplexer Systeme zu erreichen, brauchen Programmierer ein transparentes Effizienzmodell der Programmiersprache. Die Sprachkonstrukte des Objektsystems müssen daher das Verursacherprinzip einhalten und dieses dem Programmierer bewußt machen. Schließlich muß das Sprachdesign verschiedene Implementierungstechniken unterstützen, um eine umfassende Effizienz zu ermöglichen.

Auf der Grundlage dieser Kriterien und der bisherigen Objektsysteme habe ich objektorientierte *Sprachkonzepte* entwickelt, die zum einen bessere Ausdrucksmöglichkeiten schaffen und zum anderen effizienter implementierbar sind, z. B. im Vergleich zu CLOS. Mit dem Entwurf von TELOS wurden sie konkretisiert.

Objektorientierte Sprachkonzepte lassen sich unter vier Aspekten diskutieren: der *Objektstruktur*, dem *Objektverhalten*, der *Vererbung* und der *Reflektion*. Die ersten drei Aspekte können in einer Ebene behandelt werden, der sogenannten *Objektebene*. Die Objektstruktur und das Objektverhalten sind der Gegenstand von Vererbungskonzepten. Mit der Reflektion wird eine weitere Ebene betreten, die sogenannte *Metaobjektebene*. Zusammen erhält man eine metareflektive Systemarchitektur. Auf der Metaobjektebene werden Metaobjekte zu Objekten erster Ordnung. Wie man auf der Objektebene seine Anwendungsklassen erweitert und spezialisiert, so kann man auf der Metaobjektebene das Objektsystem der Programmiersprache erweitern und spezialisieren. Diese neue Dimension objektorientierter Programmierung wurde bisher nur in CLOS richtig unterstützt. Irrtümlicherweise glaubte man bisher, als Voraussetzung dafür die Quellcodeinterpretation bzw. -kompilation zur Laufzeit zu benötigen. Mit dieser Arbeit habe ich erstmals nachgewiesen, daß dies nicht der Fall ist. Damit sind die Grundlagen geschaffen worden, um die Konzepte der Spracherweiterungsprotokolle aus TELOS auch in traditionellen, compilerorientierten Programmiersprachen, wie Ada, Pascal, Eiffel, C++ und natürlich JAVA, zu unterstützen. Dazu ist lediglich eine Ergänzung des Laufzeitsystems dieser Sprachen um einen kleinen Metaobjektkern mit Objekten und Operationen erforderlich. Der Compiler kann dann den entsprechenden Code aus Anwendungen der Metaobjektprogrammierung erzeugen, der sich auf den Kern des Laufzeitsystems stützt. Einige der hier vorgestellten Metaobjektprotokolle setzen funktionale Objekte erster Ordnung voraus, wie sie in allen funktionalen Sprachen vorhanden sind. Aber selbst darauf kann im Prinzip verzichtet werden.

Die *Klassifikation der Struktur und des Verhaltens* von Objekten steht im Mittelpunkt objektorientierter Programmierung. Mit der Unterstützung des *Generalisierens und Spezialisierens* von Objektklassen wird die Hauptaufgabe eines Objektsystems angesprochen:

- Um die Objektstruktur spezialisieren zu können, müssen die Felder der Superklassen ererbt und ihre Annotationen wie z. B. der Default-Wert spezialisiert werden können. Dies ist weder in Smalltalk noch in C++ noch in JAVA möglich. In diesen

Sprachen können nur neue Felder definiert werden, ererbte Felder werden unspezialisiert übernommen.

- Um das Objektverhalten spezialisieren zu können, müssen Methoden der Superklassen ererbt und speziellere Methoden definiert werden können. Dabei muß es möglich sein, die ererbte Methode auf abstrakte Weise wiederzuverwenden und zu ergänzen. Dazu braucht man mindestens die einfache (explizite) Methodenkombination mit `callNextMethod` als die Generalisierung von `super()` aus Smalltalk sowie `inner` aus BETA. Dies ist in C++ nicht gegeben. Im Unterschied zu CLOS unterscheide ich in ΤΕΛΟΣ zwischen *expliziter und impliziter Methodenkombination*, um z. B. JAVA-Konstruktoren nicht als Einzelfall, sondern als Methoden mit impliziter Methodenkombination zu subsumieren.

Die *Vererbung* regelt, wie das Generalisieren und Spezialisieren von statten geht. Als ein wesentliches Resultat dieser Arbeit wurde die Mixin-Vererbung entwickelt, um die Nachteile der einfachen und (allgemeinen) multiplen Vererbung zu überwinden. Dabei wird zwischen wesentlichen und zusätzlichen Aspekten von Objekten unterschieden. Letztere können mehrfach ererbt werden, in Analogie zur Aneinanderreihung mehrerer Adjektive in der natürlichen Sprache:

- *Basisklassen*, die wesentliche Aspekte repräsentieren, bilden untereinander eine einfache Vererbungshierarchie.
- *Mixin-Klassen*, die zusätzliche Aspekte repräsentieren, können mehrfach geerbt werden. Allerdings kann *eine* Basisklasse nur *disjunkte* Mixin-Klassen erben.

Effiziente Implementierungen von Objektsystemen setzen ein entsprechendes Sprachdesign voraus. Dieses sollte idealerweise fünf Implementierungstechniken unterstützen:

- Modul-Kompilation,
- Applikations-Kompilation,
- inkrementelle Kompilation,
- Bytecode-Kompilation bzw. -Interpretation und
- Einbettungstechnik.

Dabei kann die Bytecode-Kompilation auch unter der Modul-Kompilation subsumiert werden. Außerdem profitiert die Einbettungstechnik von allen Formen der Kompilation der Basissprache. Daß auch Sprachen der Lisp-Familie von der Modul- und Applikations-Kompilation profitieren können, wurde im Projekt APPLY nachgewiesen. Der dafür notwendige Verzicht auf Quellcodeinterpretation und -kompilation zur Laufzeit reicht im Prinzip aus, um traditionelle Kompilationstechniken erfolgreich anzuwenden. Im Verzicht auf Quellcodeinterpretation liegt auch der Schlüssel für die Effizienz meiner Implementierung.

Im Unterschied zu CLOS, daß mit seinen Redefinitionsmöglichkeiten primär auf inkrementelle Kompilation ausgerichtet ist, unterstützt ΤΕΛΟΣ alle genannten Implementierungstechniken. Insbesondere trägt die strikte Unterscheidung zwischen Sprachkonstrukten der

Objekt- und der Metaobjektebene sowie die Modularisierung des Metaobjektprotokolls dazu bei, auch die Modul- und Applikations-Kompilation zu unterstützen, was ein Novum für metareflektive Objektsysteme darstellt.

Aber selbst mit der Einbettungstechnik und ohne explizite Verwendung des inkrementellen Compilers ist meine TELOS-Implementierung insgesamt effizienter als die derzeit besten kommerziellen CLOS-Realisierungen. Dies ist das Resultat eines konsequenten Sprachdesigns und einer einfachen Implementierungstechnik unter Beachtung des Verursacherprinzips. Dies hat insbesondere am Beispiel des *Redefinitionskonzepts* in TELOS dazu beigetragen, den entsprechenden Overhead auf Objekte zu beschränken, die diese Funktionalität auch wirklich benötigen. Darüberhinaus konnte die *Objekterzeugung* und die *Methodenkombination* in TELOS wesentlich effizienter realisiert werden als in CLOS.

Ausblick

Natürlich konnten in dieser Arbeit nicht alle Aspekte objektorientierter Programmiersprachen in der gewünschten Tiefe behandelt werden. Bezüge zu deklarativen Programmiersprachen blieben unberücksichtigt. Fragen der Parallelität und Verteilung von Objekten wurden nur oberflächlich angesprochen.

Aber auch näherliegende und auf den erzielten Ergebnissen dieser Arbeit aufsetzende Fragestellungen könnten unmittelbar aufgegriffen werden. Dazu zählt die Erweiterung des Metaobjektprotokolls bezogen auf das Modulkonzept der Objektebene. Das präsentierte Metaobjektprotokoll reflektiert hauptsächlich Aspekte des Objektsystems selbst, nicht aber die des Modulkonzepts. Wenn man an die Realisierung wissensbasierter Systeme denkt, stellt sich die Frage nach der Modularisierung von Wissensbasen. Will man diese auf Modularisierungsmittel der Programmiersprache abbilden, so braucht man ein entsprechendes Introspektions-, Erweiterungs- und Spezialisierungsprotokoll bezüglich der Import- und Exportbeziehungen zwischen Modulen.

Da Module im wesentlichen auch Kompilationseiheiten sind, beginnt hier die Schnittstelle eines Modulkompilers. Natürlich müßte dabei strikt darauf geachtet werden, daß Sprachkonstrukte der Objekt- und der Metaobjektebene auseinander gehalten werden. In diesem Zusammenhang könnten erste Ansätze sogenannter *Compilezeit-MOPs* systematisch verfolgt werden. Letztere können dazu führen, daß die meisten Operationen der Metaobjektebene zur Übersetzungszeit ausgeführt werden. Hierzu können die Techniken der partiellen Auswertung von Ausdrücken (engl. *partial evaluation*) verwendet werden.

Im Zusammenhang mit der konkreten Übertragung der Ergebnisse dieser Arbeit auf compiler-orientierte Sprachen wie JAVA stellt sich die Frage, inwieweit die Typkorrektheit von Programmen zur Übersetzungszeit geprüft werden kann. Insbesondere wirft die Spezialisierung von Feldern bzgl. der Typannotation im Kontext multipler Vererbung einige Probleme auf, weil sie möglicherweise eine neue Klasse induziert, die im zu übersetzenden Modul nicht explizit definiert wird. In diesem Zusammenhang muß der generellen Frage nachgegangen werden, wie Typkorrektheit mit generischen Programbausteinen ohne Effizienzverlust vereinbart werden kann.

Bezogen auf effiziente Implementierungen wäre die Konstruktion eines optimierenden Modul- und Applikations-Compilers für EULISP zu nennen. Leider konnten die im Rahmen des APPLY-Projekts hierzu begonnenen Arbeiten nicht fortgesetzt werden, waren

doch die bereits erzielten Resultate sehr vielversprechend. Im Prinzip konnte für EuLisp die gleiche Laufzeit- und Speichereffizienz wie für C++ erreicht werden. Es wäre sehr wünschenswert, dieses Ergebnis auch praktisch nutzbar zu machen. Da offensichtlich nicht damit zu rechnen ist, daß Lisp in der nächsten Zeit eine vergleichbare Rolle wie C++ oder Java spielen wird, wäre es sinnvoll für die entsprechnde Interoperabilität zu sorgen. Dann könnten die Vorteile beider Welten miteinander genutzt werden, ohne jeweils alles nachziehen zu müssen, was die andere Welt einem voraus hat. Ein praktischer Weg könnte hier darin bestehen, aus EuLisp-Modulen Java-Bytecode zu generieren, was für Scheme bereits gezeigt wurde.

Anhang A

Glossar

Abstrakte Klasse:
Eine abstrakte Klasse kann selbst nicht instanziiert werden. Sie dient zur Begriffsbildung und Strukturierung der Klassenhierarchie.

Attribut:
Siehe Feld.

Classifier:
Ein Classifier ermittelt die Subsumtions- bzw. Superkonzeptbeziehungen für eine Menge von Konzeptbeschreibungen und findet für ein Objekt das speziellste Konzept, dessen charakteristische Merkmale es erfüllt.

change-class:
Eine Objektsystem-Operation, um ein Objekt einer anderen (Repräsentations-)Klasse zuzuordnen als der bei seiner Erzeugung spezifizierten.

Differenzieren von Klassen:
Vom Differenzieren einer Klasse spricht man, wenn zwei oder mehrere Subklassen einer Klasse zur expliziten Unterscheidung bzgl. eines Kriteriums gebildet werden. Differenzieren bedeutet „gegeneinander abgrenzen“. Differenzierende Klassen müssen daher disjunkt sein und dürfen bei multipler Vererbung keine gemeinsame Subklasse haben.

Direkte Instanz:
Eine direkte Instanz einer Klasse C_1 ist eine Instanz, die keiner anderen Klasse gehört, die spezieller als C_1 ist.

Direkte Subklasse:
Die Klasse C_2 ist eine direkte Subklasse von C_1, wenn C_1 direkte Superklasse von C_2 ist.

Direkte Superklasse:
Die Klasse C_1 ist eine direkte Superklasse von C_2, wenn C_1 Superklasse von C_2 ist, die beiden Klassen nicht identisch sind und keine andere Klasse existiert, die Superklasse von C_2 und Subklasse von C_1 ist. Das heißt, in der Klassenhierarchie existiert keine weitere Klasse zwischen C_1 und C_2. Oder anders ausgedrückt: Eine Klasse C_2 nennt in ihrer Definition C_1 explizit als Superklasse.

Generalisieren:
Generalisieren ist der inverse Schritt zum Spezialisieren. Die Struktur und das Verhalten

von Objekten werden generealisiert, indem Superklassen eingeführt werden, in denen ähnliche oder gleiche Felder und Methoden mehrerer Klassen gebündelt werden. Siehe auch Spezialisieren.

Generische Funktion:
Eine Funktionen heißt generisch, wenn sie für unterschiedliche Argumenttypen verschiedene Methoden besitzt, die das Verhalten der generischen Funktion spezifizieren. Die Signaturen der Methoden müssen die Signatur der generischen Funktion spezialisieren. Beim Aufruf einer generischen Funktion wird in der Regel die speziellste Methode anhand der Typen der aktuellen Parameter bestimmt und ausgeführt.

Feld:
Die Werte der Felder eines Objekts repräsentieren seinen Zustand. Die Felder (Instanzvariablen, Slots, Merkmale, Attribute etc.) werden von der Klasse eines Objekts definiert, die dadurch die Struktur ihrer Instanzen festlegt.

Indirekte Instanz:
Eine indirekte Instanz einer Klasse C_1 ist eine Instanz, deren Klasse indirekte Subklasse von C_1 ist.

Indirekte Subklasse:
Eine Klasse C_3 ist indirekte Subklasse von C_1, wenn C_3 Subklasse von C_1 ist, die beiden Klassen nicht identisch sind und mindestens eine Klasse C_2 existiert, die Superklasse von C_3 ist und Subklasse von C_1. Das heißt, in der Klassenhierarchie existiert noch mindestens eine Klasse zwischen C_3 und C_1.

Indirekte Superklasse:
Eine Klasse C_3 ist indirekte Superklasse von C_1, wenn C_3 Superklasse von C_1 ist, die beiden Klassen nicht identisch sind und mindestens eine Klasse C_2 existiert, die Subklasse von C_3 ist und Superklasse von C_1. Das heißt, in der Klassenhierarchie existiert noch mindestens eine Klasse zwischen C_3 und C_1.

Initialisierung von Objekten:
Die Erzeugung von Objekten erfolgt in der Regel in zwei Schritten: der Allozierung und Initialisierung. Im zweiten Schritt werden die allozierten Felder eines Objekts mir initialen Werten belegt.

Instanz:
Instanzen sind Ausprägungen von Klassen. Diese können terminal, d. h. Blattknoten in der Instanziierungshierarchie, sein, aber sie können auch Klassen darstellen.

Instanziierung:
Erzeugung eines Objekts als Instanz einer Klasse.

Instanziierungsgraph
Instanziierungshierarchie:
Klassen und Instanzen bilden aufgrund der Instanz-Beziehung eine Hierarchie, die durch einen Graphen veranschaulicht werden kann.

Instanzvariable:
Siehe Feld.

Klasse:
Klassen beschreiben die Struktur (Felder) und das Verhalten (Methoden) ihrer Instanzen.

Wenn eine Klasse keine Metaklasse darstellt, sind ihre Instanzen terminal, d. h. nicht instanziierbar.

Klassenhierarchie:
Mit Klassenhierarchie ist in der Regel die Hierarchie der Superklassenbeziehung gemeint. Diese läßt sich meist durch einen gerichteten, azyklischen Graphen veranschaulichen. Klassen können aber noch weitere Hierarchien bilden, z. B. die Aggregations- bzw. Teil-/Ganzeshierarchie, deren induzierter Graph auch zyklisch sein darf. Siehe auch Objekthierarchie.

Klassenvariable:
Eine Klassenvariable ist ein Feld der Klasse einer Instanz, im Gegensatz zu Instanzvariablen als Felder der Instanz selbst. Da Klassen in ΤΕΛΟΣ auch Instanzen sind, macht sich der Unterschied nur im eingenommenen Standpunkt bemerkbar. Es gibt keine Klassenvariablen an sich, es sind die Felder eines Klassenobjekts aus der Sicht seiner Instanzen.
Siehe auch Feld.

Message Passing:
Siehe Versenden von Nachrichten.

Metaklasse:
Wie alle Klassen beschreiben Metaklassen die Struktur und das Verhalten ihrer Instanzen, die jedoch selbst Klassen sind.

Metaobjektprotokoll:
Ein Metaobjektprotokoll öffnet die Implemetierung einer objektorientierten Programmiersprache auf eine kontrollierte Weise, so daß der Programmierer die Sprache den Anforderungen seiner Applikation anpassen kann, anstatt die Applikation auf die Gegegenheiten einer Sprache auszurichten. Der Programmierer erhält die Möglichkeit, Systemklassen und Systemfunktionalität zu spezialisieren und somit wiederzuverwenden. Dazu müssen die Klassen Objekte erster Ordnung sein.

Methode:
Eine Methode beschreibt eine Prozedur bzw. Funktion, die für eine Klasse definiert wird und für ihre Instanzen anwendbar ist. Die Menge aller Methoden einer Klasse wird als ihr Protokoll bezeichnet. **Multimethoden** beschreiben das Verhalten mehrerer Klassen in ihrem Zusammenwirken.

Mixin-Klasse:
Mixin-Klassen beschreiben ergänzende Eigenschaften von Objekten, die von Basisklassen ererbt werden können. Mixin-Klassen sind abstrakt und besitzen keine direkten Instanzen. Sie stellen ein Mittel zur besseren Strukturierung und Modellierung einer Domäne dar.

Objekt:
Objekt ist der Oberbegriff für alle objekt-orientiert verwendbaren Entitäten in einem Programm: Instanzen, Klassen und Metaklassen. In ΤΕΛΟΣ ist alles ein Objekt, das als Wert einer Variable fungieren kann. In JAVA gibt es primitive Werte, z. B. vom Typ `int`, `boolean` etc., die keine Objekte sind. In klassenbasierten Sprachen ist jedes Objekt Instanz einer Klasse.

Objekte erster Ordnung:
Ein Objekt erster Ordnung kann in einer Programmiersprache dynamisch erzeugt, an Variablen oder Konstanten gebunden, in Datenstrukturen gespeichert und als Argument

bzw. als Resultat von Funktionen verwendet werden. In EuLisp sind sowohl Funktionen als auch Klassen, Felder, generische Funktionen und Methoden Objekte erster Ordnung, jedoch nicht die Module und Variablenbindungen.

Objekthierarchie:
Objektklassen bilden aufgrund der Typannotationen ihrer Felder eine Hierarchie, die durch einen Graphen veranschaulicht werden kann. Dieser kann auch zyklisch sein. Ist der Graph nicht zyklisch, kann er auch die Teil-/Ganzes-Beziehung ausdrücken.

Signatur:
Die Signatur einer generischen Funktion, Methode oder Relation legt die Typen ihrer Argumente und ggf. ihres Ergebnisses fest. Eine Signatur spezialisiert eine andere, wenn jedes Element der spezielleren Signatur ein Subtyp des entsprechenden Element der allgemeineren Signatur ist.

Slot:
Siehe Feld.

Spezialisierung von Klassen:
Eine Klasse C_2 spezialisiert eine Klasse C_1, wenn C_2 eine Subklasse von C_1 ist und C_2 neue Eigenschaften zu denen von C_1 hinzufügt oder bereits existierende Eigenschaften spezialisiert.

Spezialisierung von Struktur:
Eine Klasse C_2 spezialisiert die Struktur einer Klasse C_1, wenn C_2 eine Subklasse von C_1 ist und C_2 neue Felder zu denen von C_1 hinzufügt oder bereits existierende Felder spezialisiert, indem die Feldannotationen speziellere Werte erhalten. Dies betrifft vor allem die Typannotation.

Spezialisierung von Verhalten:
Eine Klasse C_2 spezialisiert das Verhalten einer Klasse C_1, wenn C_2 eine Subklasse von C_1 ist und C_2 neue Operationen zu denen von C_1 hinzufügt oder für bereits existierende Operationen neue Methoden definiert. Diese können die allgemeineren Methoden z. B. über `callNextMethod` aufrufen.

Subklasse:
Das Verhalten und die Struktur einer Klasse C_1 kann von anderen Klassen, ihren Subklassen, direkt oder indirekt ererbt werden. Als Konvention wird eine Klasse in die Menge ihrer Subklassen aufgenommen, um in vielen Kontexten sie selbst nicht zusätzlich nennen zu müssen. Eine Subklasse kann entweder direkt oder indirekt sein.

Subsumtion:
Ein Begriff aus dem Kontext von Konzeptsprachen. Ein Konzept C_1 subsumiert ein Konzept C_2 genau dann, wenn unter allen möglichen Interpretationen alle Instanzen von C_2 auch Instanzen von C_1 sind.

Superklasse:
Eine Klasse C_1 ist Superklasse von C_2, wenn C_2 direkte oder indirekte Subklasse von C_1 ist.

Vererbung:
Eine Metapher die zum einen an biologische Abstammung und zum anderen an das Erbrecht anknüpft. Vererbung soll auch an die Klassifikation von Pflanzen und Lebewesen erinnern. Im Kontext von objektorientierten Programmiersprachen soll Vererbung

zum Ausdruck bringen, daß ein Objekt nicht von Grund auf neu erzeugt wird, sondern auf der Basis von anderen Objekten. Dabei übernimm das neue Objekt die Eigenschaften von anderen Objekten, spezialisiert sie ggf. und fügt neue hinzu. Dadurch können Redundanzen im Programmcode vermieden werden. Die Vererbung unterstützt somit die Wiederverwendung bereits existierenden Codes. In zweistufigen Objektsystemen bezieht sich Vererbung auf Klassen und beschreibt die Ober- und Unterklassenbeziehungen.

Versenden von Nachrichten:
Eine Metapher aus dem Kommunikationsmodell von Smalltalk. Objekte kommunizieren miteinander durch das Versenden von Nachrichten. Ein Objekt reagiert auf eine Nachricht, indem es eine Methode ausführt. Die Metapher suggeriert asynchrone Kommunikation zwischen Objekten und sollte auch nur dafür verwendet werden. In einigen Sprachen ist das Versenden von Nachrichten als Standardfall (wenn Parallelität keine Rolle spielt) durch den Aufruf von generischen Funktionen ersetzt worden.

Wertebereichsbeschränkung:
Sie legt den Ergebnistyp einer Funktion fest.

Anhang B

Performanzmessungen von MCS 0.5

B.1 Erster Vergleichstest

Folgende Tabelle zeigt die vollständigen Ergebnisse des ersten Vergleichstests aus [Walther, 1988].

Test	Beschreibung	WFS	MCF	PCL	MCS
ft1	defflavor 15	6,650	7,917	21,817	6,000
ft2	make-instance 80	5,833	79,167	1,333	0,167
ft3	:read 3200	136,480	40,733	13,167	5,650
ft4	:write 3200	157,883	118,150	14,867	2,767
ft5	:send-if-handles 100	23,867	23,35	3,533	4,733
ft6	compile-flavor-methods	26,067	34,833	-	-
ft3c	:read 3200	70,700	13,200	12,733	5,483
ft4c	:write 3200	85,517	27,167	14,900	2,767
ft5c	:send-if-handles 100	26,233	11,217	3,533	4,733

Tabelle B.1: Ausführungszeiten auf Mac II, Allegro CL in Sekunden

B.2 Zweiter Vergleichstest

Folgende Tabellen zeigen die vollständigen Einzelergebnisse des von mir durchgeführten Vergleichstests für den Apple Macintosh II und die SUN Workstation.

Test	Beschreibung	PCL	MCF	MCS
ft1	defflavor 15	29,000	7,483	6,583
ft2	make-instance 80	3,830	40,000	0,500
ft3	:read 3200	1,083	16,617	1,383
ft3c	:read 3200	0,667	2,867	1,383
ft3s	get-slot 3200	1,600	0,383	0,217
ft4	:write 3200	1,117	55,667	1,467
ft4c	:write 3200	0,633	2,733	1,467
ft4s	set-slot 3200	0,633	2,733	1,467
ft5	:send-if-handles 100	2,700	29,650	3,017
ft5c	:send-if-handles 100	2,700	17,267	1,933
ft6	compile-flavor-methods	-	91,767	-

Tabelle B.2: Ausführungszeiten auf Macintosh II, Allegro CL in Sekunden

Test	Beschreibung	Flavors	MCF	MCS
ft1	defflavor 15	5,040	2,600	18,940
ft2	make-instance 80	9,020	1,090	0,040
ft3	:read 3200	0,980	10,660	1,200
ft3c	:read 3200	0,900	3,380	1,200
ft4	:write 3200	1,020	57,580	1,280
ft4c	:write 3200	0,960	4,420	1,280
ft5	:send-if-handles 100	1,440	27,160	2,820
ft5c	:send-if-handles 100	1,420	9,640	2,280
ft6	compile-flavor-methods	0,020	27,000	-

Tabelle B.3: Ausführungszeiten auf SUN, Lucid CL in Sekunden

B.3 Vergleichstest für BABYLON-Wissensbasen

Die folgende Tabelle zeigt die vollständigen Meßergebnisse des Vergleichstests der BABYLON-Prozessoren für konkrete Wissensbasen.

Test	Beschreibung	MCF	MCS
frames1	Alle Slots aller Instanzen, Wissensbasis Pilze	29,617	5,583
frames1c	frames1 nach Kompilation	6,983	5,583
frames2	100 Slotzugriffe Wissensbasis Pilze	4,433	1,333
frames2c	frames3 nach Kompilation	2,633	1,333
frames3	100 Slotzugriffe Wissensbasis Urlaub	8,267	1,433
frames3c	frames2 nach Kompilation	2,050	1,433
rules1	50 Regeln	171,767	18,050
rules1a	50 Regeln, neuer Algorithmus	171,767	14,000
rules2	200 Regeln	2545,783	268,650
rules2a	200 Regeln, neuer Algorithmus	2545,783	208,067
prolog1	naive reverse, 30 Elemente, sec	1325,467	10,033
prolog2	naive reverse, 30 Elemente, lips	0,430	49,430

Tabelle B.4: Vergleich der Ausführungszeiten von BABYLON-Prozessoren in Sekunden

Prolog-Test	MCF	MCS	WAM
naive reverse, 30 Elemente	1325,467	10,033	5,417

Tabelle B.5: BABYLON-Prolog-Prozessor im Vergleich zur WAM in Sekunden.

Prolog-Test	MCF	MCS	WAM
naive reverse, 30 Elemente	0,43	49,43	91,5

Tabelle B.6: BABYLON-Prolog-Prozessor im Vergleich zur WAM in lips.

Anhang C

Performanzmessungen von MCS 1.0

Die vollständigen Meßergebnisse von MCS 1.0:

Test	Beschreibung	Allegro CL Version 3		Allegro CL Version 4.0	
		PCL	MCS	CLOS	MCS
	Objekterzeugung				
t1	defclass 10 x	5117	750	2549	584
t2	defmethod 100 x	1533	134	567	117
t3	make 1000 x	2917	334	533	267
	Feldzugriffe				
t4	slot-value 10000 x	416	216	283	200
t5	reader 10000 x	183	200	134	183
t6	writer 10000 x	200	200	183	183
	Generischer Dispatch				
t7	f(x) 10000 x	-	-	-	-
t8	gf(x) 10000 x	200	200	167	167
t9	gf(x,y) 10000 x	283	250	217	200
	Methodenkombination				
t5a	reader 10000 x	183	200	134	183
t5b	reader + :before 10000 x	1267	733	516	767
t5c	reader + ...:after 10000 x	1450	1100	583	983
t5d	reader + ...:around 10000 x	3083	1267	1416	1184
t8a	gf(x) 10000 x	200	200	167	167
t8b	gf(x) + :before 10000 x	917	767	217	800
t8c	gf(x) + ...:after 10000 x	1100	883	250	884
t8d	gf(x) + ...:around 10000 x	2683	1083	967	1017

Tabelle C.1: Ausführungszeiten auf Sun Workstation, Franz Allegro Common Lisp (ACL) in Millisekunden

Speicherbedarf	PCL	MCS
Quellcode	1,185	0,294
Binärcode (RISC)	1,143	0,652

Tabelle C.2: Speicherbedarf von PCL und MCS in Franz Allegro CL in Megabytes

Anhang D

Performanzmessungen von TELOS

D.1 Vergleichstest auf Sun Workstation

Die Tests wurden auf einer Sparcstation Ultra 1 (Modell 140) mit einem Ultra-Sparc-Prozessor (143 Mhz) und 272MB RAM unter SunOS 5.5.1 (Solaris) in Allegro Commonlisp Version 4.3.1 [SPARC; R1] durchgeführt.

D.2 Vergleichstest auf Apple Macintosh

Die Tests wurden auf einem Apple Macintosh PowerBook Duo280c mit einem PowerPC-Prozessor und 20MB RAM unter MacOS 7.5.5 in Macintosh CommonLisp Version 4.1 durchgeführt.

Test	Beschreibung	CLOS	TELOS
Objekterzeugung			
t1	defclass 10 x	57	10
t2	defmethod 100 x	110	20
t3	make 1000 x	140	21
t3a	static make 1000 x	4	21
Feldzugriffe			
t4	slot-value 10000 x	33	55
t4a	static slot-value 10000 x	26	66
t5	reader 10000 x	24	23
t6	writer 10000 x	23	22
Generischer Dispatch			
t7	f(x) 10000 x	23	23
t8	gf(x) 10000 x	27	34
t9	gf(x,y) 10000 x	37	44
Methodenkombination			
t5a	reader 10000 x	24	23
t5b	reader + :before 10000 x	32	51
t5c	reader + ...:after 10000 x	42	67
t5d	reader + ...:around 10000 x	86	85
t8a	gf(x) 10000 x	27	34
t8b	gf(x) + :before 10000 x	27	44
t8c	gf(x) + ...:after 10000 x	31	66
t8d	gf(x) + ...:around 10000 x	75	76

Tabelle D.1: Ausführungszeiten auf einer Sun Workstation in Franz Allegro CL 4.3.1 in Millisekunden.

Test	Beschreibung	CLOS		TELOS	
		cons-Zellen	Bytes	cons-Zellen	Bytes
Objekterzeugung					
t1	defclass 10 x	6969	20864	1131	19880
t2	defmethod 100 x	1147	2872	34	2976
t3	make 1000 x	10223	64192	3000	64000
t3a	static make 1000 x	0	64000	3000	64000
Feldzugriffe					
t4	slot-value 10000 x	2	0	2	0
t4a	static slot-value 10000 x	42	320	6	24
t5	reader 10000 x	26	160	1	0
t6	writer 10000 x	28	160	2	0
Generischer Dispatch					
t7	f(x) 10000 x	1	0	1	0
t8	gf(x) 10000 x	32	184	6	24
t9	gf(x,y) 10000 x	42	216	10	24
Methodenkombination					
t5a	reader 10000 x	26	160	1	0
t5b	reader + :before 10000 x	423	624	13	24
t5c	reader + ...:after 10000 x	3226	24936	14	24
t5d	reader + ...:around 10000 x	15477	278504	26	24
t8a	gf(x) 10000 x	32	184	6	24
t8b	gf(x) + :before 10000 x	409	424	13	24
t8c	gf(x) + ...:after 10000 x	537	424	14	24
t8d	gf(x) + ...:around 10000 x	100981	2400432	26	24

Tabelle D.2: Speicherbedarf auf einer Sun Workstation, Franz Allegro CL 4.3.1 in der Anzahl der cons-Zellen und weiteren Bytes.

Test	Beschreibung	CLOS	MCS 0.5	MCS 1.0	TELOS	CL-TELOS
Objekterzeugung						
t1	defclass 10 x	504	1425	374	143	536
t2	defmethod 100 x	1430	250	1480	110	140
t3	make 1000 x	213	34	144	58	1084
t3a	static make 1000 x	138	29	156	61	1063
Feldzugriffe						
t4	slot-value 10000 x	30	23	48	63	181
t4a	static slot-value 10000 x	51	83	84	136	280
t5	reader 10000 x	32	73	45	20	14
t6	writer 10000 x	36	110	48	17	16
Generischer Dispatch						
t7	f(x) 10000 x	10	10	10	10	10
t8	gf(x) 10000 x	36	90	45	45	118
t9	gf(x,y) 10000 x	80	180	68	75	307
Methodenkombination						
t5a	reader 10000 x	32	73	45	20	14
t5b	reader + :before 10000 x	152	100	77	72	145
t5c	reader + ...:after 10000 x	207	148	92	90	200
t5d	reader + ...:around 10000 x	313	156	99	100	218
t8a	gf(x) 10000 x	36	90	45	45	118
t8b	gf(x) + :before 10000 x	122	86	81	59	132
t8c	gf(x) + ...:after 10000 x	152	113	86	66	206
t8d	gf(x) + ...:around 10000 x	230	232	163	77	209

Tabelle D.3: Ausführungszeiten auf einem Apple Macintosh PB 280 in Macintosh CL 4.1 in Millisekunden.

Test	Beschreibung	CLOS	MCS 0.5	MCS 1.0	TELOS	CL-TELOS
Objekterzeugung						
t1	defclass 10 x	41944	206040	58848	19352	71248
t2	defmethod 100 x	104560	5600	145520	15040	10440
t3	make 1000 x	40024	128024	112024	64024	728024
t3a	static make 1000 x	40000	112000	112000	64000	728024
Feldzugriffe						
t4	slot-value 10000 x	16	16	16	16	16
t4a	static slot-value 10000 x	32	160	136	64	160088
t5	reader 10000 x	16	112	112	8	8
t6	writer 10000 x	24	80120	144	16	8
Generischer Dispatch						
t7	f1(x) 10000 x	1	1	1	1	8
t8	gf(x) 10000 x	4	14	12	6	160008
t9	gf(x,y) Parameter 10000 x	14	8014	18	10	320013
Methodenkombination						
t5a	reader 10000 x	16	112	112	8	8
t5b	reader + :before 10000 x	88	144	80208	72	160104
t5c	reader + ...:after 10000 x	104	160	80224	80	160120
t5d	reader + ...:around 10000 x	120	80168	80256	88	160136
t8a	gf(x) 10000 x	4	136	12	6	160008
t8b	gf(x) + :before 10000 x	8	144	80020	7	160010
t8c	gf(x) + ...:after 10000 x	10	160	80022	8	160013
t8d	gf(x) + ...:around 10000 x	12	80144	80026	10	160014

Tabelle D.4: Speicherbedarf auf einem Apple Macintosh PB 280 in Macintosh CL 4.1 in Bytes.

Anhang E

Implementierung der Mixin-Vererbung in TELOS

Folgendes Modul zeigt, wie man die Mixin-Vererbung durch Spezialisieren der Metaobjektklassen und unter Ausnutzung des Metaobjektprotokolls von ΤΕΛΟΣ einfach implementieren kann. Konkret wurde hier meine ΤΕΛΟΣ-Implementierung in CommonLisp Celos verwendet.

```
;;; -*- Mode: Lisp; Syntax: Common-Lisp; Base: 10; Package: "TELOS" -*-
;;; ------------------------------------------------------------------------
;;; EuLisp Module: mixin inheritance                    copyright 1993 by GMD
;;; Author:        Harry Bretthauer
;;; Date:          13.09.93
;;; ------------------------------------------------------------------------

(in-package "TELOS")
;#-CLICC(in-package "TELOS")
;#+CLICC(in-package "TELOS" :use ())

(defmodule mixins
  ;; interface:
  (import (eulisp-level-0
           (only (<object>
                  <class>
                  <simple-class>
                  <local-slot>
                  compute-class-precedence-list
                  compatible-superclasses-p
                  compatible-superclass-p
                  compute-inherited-keywords
                  compute-keywords
                  compute-inherited-slots
                  compute-slots
```

```
                compute-specialized-slot
                compute-slot-reader
                compute-slot-writer
                ensure-slot-reader
                ensure-slot-writer) celos-classes))

 syntax (only (defclass
               defcondition
               defmethod) celos-syntax)

 export (<base-class>
         <mixin-class>
         defmixin
         defbase))

;; implementation:

(defmacro defmixin (name supers slots . options)
  `(defclass ,name ,supers ,slots ,@options class <mixin-class>))

(defmacro defbase (name supers slots . options)
  `(defclass ,name ,supers ,slots ,@options class <base-class>))

(defclass <common-class> (<class>) () abstract t)
(defclass <base-class> (<common-class>) ())
(defclass <mixin-class> (<common-class>) () predicate mixin-class-p)

(defmethod compatible-superclasses-p ((cl <base-class>)
                                      (direct-superclasses <cons>))
  (labels ((loop (superclasses)
                 (if (null (rest superclasses))
                   (compatible-superclass-p cl (first superclasses))
                   (and (mixin-class-p (first superclasses))
                        (compatible-superclass-p cl (first superclasses))
                        (loop (rest superclasses))))))
    (loop direct-superclasses)))

(defmethod compatible-superclasses-p ((cl <mixin-class>)
                                      (direct-superclasses <cons>))
  (labels ((loop (superclasses)
                 (cond
                     ((null (rest superclasses))
                      (compatible-superclass-p cl (first superclasses)))
                   ((mixin-class-p (first superclasses))
                    (loop (rest superclasses)))
                   (t false))))
```

```
    (loop direct-superclasses)))

(defmethod compatible-superclass-p ((cl <mixin-class>) (superclass <class>))
  (declare (ignore cl))
  (or (mixin-class-p superclass)
      (eq superclass <object>)))

;;; Some methods must be defined for both, <base-class> and
;;; <mixin-class>. Thus we define them for a common superclass <common-class>.

(defmethod compute-class-precedence-list ((cl <common-class>)
                                          (direct-superclasses <list>))
  ;; Returns the class-precedence-list, if traversing is successful.
    (cons cl (remove-duplicates-from-end
              (apply append
                     (mapcar class-precedence-list direct-superclasses)))))

(defun legal-join-node-p (node) (eq node <object>))

(defun remove-duplicates-from-end (elements)
  (labels ((fold (elements result)
 (cond
  ((null elements) result)
  ((memq (car elements) result)
                   (if (legal-join-node-p (car elements))
                     (fold (cdr elements) result)
                     (error "Illegal join node detected."
                            <illegal-inheritance-hierarchy>
                            elements)))
  (t (fold (cdr elements) (cons (car elements) result))))))
  (fold (reverse elements) '())))

(defmethod compute-inherited-keywords ((cl <common-class>)
                                       (direct-superclasses <cons>))
  (declare (ignore cl))
   (mapcar class-keywords direct-superclasses))

(defmethod compute-keywords ((cl <common-class>) (direct-keywords <list>)
                             (inherited-keywords <list>))
  (declare (ignore cl))
   (remove-duplicates-from-end
    (apply append direct-keywords inherited-keywords)))

(defmethod compute-inherited-slots ((cl <common-class>)
                                          (direct-superclasses <list>))
   (declare (ignore cl))
```

```
    (mapcar class-slots direct-superclasses))

  (defmethod compute-slots ((cl <common-class>) (direct-sds <list>)
                            (inherited-sds-lists <list>))
    (let ((inherited-sds (apply append inherited-sds-lists)))
      ;; inherited-sds are ordered most general first
      (let ((inherited-sd-names          ; most general first
             (remove-duplicates
              (mapcar slot-name inherited-sds))))
        (let ((specialized-inherited-sds
               (mapcar (lambda (name)
                         (compute-specialized-slot
                          cl
                          (collect inherited-sds
                                   (lambda (inherited-sd)
                                     (eq name (slot-name inherited-sd)))
                                   identity)
                          (find direct-sds
                                (lambda (direct-sd)
                                  (eq name (getf direct-sd 'name)))
                                identity)))
                       inherited-sd-names))
              (new-sds
               (collect direct-sds
                        (lambda (direct-sd)
                          (not (memq (getf direct-sd 'name) inherited-sd-names)))
                        (lambda (direct-sd)
                          (compute-defined-slot cl direct-sd)))))
          (append specialized-inherited-sds new-sds)))))

  (defclass <slot-clash> (<telos-condition>) ())

  (defmethod compute-specialized-slot ((cl <common-class>)
                                       (inherited-sds <cons>)
                                       (slot-spec <list>))
    (when (> (list-length inherited-sds) 1)
      (unless (eq (slot-reader (first inherited-sds))
                  (slot-reader (second inherited-sds)))
        (error "Can't specialize different slots with the same name!"
               <slot-clash> cl inherited-sds)))
    (let ((new-sd (allocate (compute-specialized-slot-class
                             cl inherited-sds slot-spec)
                            slot-spec))
          (init-list slot-spec))
      (unless (extract-option 'default-function slot-spec)
        (let ((default-function
```

```
            (detect slot-default-function inherited-sds)))
        (when default-function
          (setq init-list `(default-function ,default-function ,@init-list)))))
    (let ((inh-keyword
           (detect slot-keyword inherited-sds))
          (new-keyword (extract-option 'keyword slot-spec)))
      (if new-keyword
        (unless (or (not inh-keyword) (eq inh-keyword new-keyword))
          (error "Illegal keyword defined." <illegal-keyword-defined>
                 cl slot-spec))
        (when inh-keyword
          (setq init-list `(keyword ,inh-keyword ,@init-list)))))
    (initialize
     new-sd
     `(name ,(slot-name (car inherited-sds))
       reader ,(slot-reader (car inherited-sds))
       writer ,(slot-writer (car inherited-sds))
       ,@init-list))))

(defun detect (fn list)
  (cond
   ((null list) false)
   ((funcall fn (car list)))
   (t (detect fn (cdr list)))))

;;; Following methods are different for efficiency reasons.
;;; Slots of mixin classes need generic accessors because their positions
;;; can be different in different ineriting base classes. Slots of base
;;; classes do not change their positions in subclasses, thus, they get
;;; non-generic accessors computed by default methods.

(defmethod compute-slot-reader ((cl <mixin-class>) (sd <local-slot>)
                                (eff-sds <cons>))
  (declare (ignore sd eff-sds))
  (generic-lambda ((obj cl))))

(defmethod compute-slot-writer ((cl <mixin-class>) (sd <local-slot>)
                                (eff-sds <cons>))
  (declare (ignore sd eff-sds))
  (generic-lambda ((obj cl) (val <object>))))

(defmethod ensure-slot-reader ((cl <base-class>) (sd <local-slot>)
                               (effective-sds <cons>)
                               (reader <function>))
  ;; the default method adds a method to the reader if it has no method
  ;; added yet. here, we need a method if it is inherited from a mixin class
```

```
  ;; too. that is the case if reader is a generic function.
  (when (generic-function-p reader)
    (let ((primitive-reader
           (compute-primitive-reader-using-slot
            sd cl effective-sds)))
      (add-method reader
                  (method-lambda ((obj cl))
                                 (funcall primitive-reader obj))))))

(defmethod ensure-slot-reader ((cl <mixin-class>) (sd <local-slot>)
                               (effective-sds <cons>)
                               (reader <generic-function>))
  ;; mixin classes have no direct instances, there is nothing to do here.
  ;; when a base class inherits a slot from a mixin class, then a method must
  ;; be added.
  (declare (ignore cl sd effective-sds))
  reader)

(defmethod ensure-slot-writer ((cl <base-class>) (sd <local-slot>)
                               (effective-sds <cons>)
                               (writer <function>))
  ;; the default method adds a method to the reader if it has no method
  ;; added yet. here, we need a method if it is inherited from a mixin class
  ;; too. that is the case if reader is a generic function.
  (when (generic-function-p writer)
    (let ((primitive-writer
           (compute-primitive-writer-using-slot
            sd cl effective-sds)))
      (add-method writer
                  (method-lambda ((obj cl) (val <object>))
                                 (funcall primitive-writer obj val))))))

(defmethod ensure-slot-writer ((cl <mixin-class>) (sd <local-slot>)
                               (effective-sds <cons>)
                               (writer <generic-function>))
  ;; mixin classes have no direct instances, there is nothing to do here.
  ;; when a base class inherits a slot from a mixin class, then a method must
  ;; be added.
  (declare (ignore cl sd effective-sds))
  writer)

(print "Mixin inheritance module has been loaded!")

)    ; end of mixins module
```

Literaturverzeichnis

[Abelson und Sussman, 1985] H. Abelson und G. J. w. J. Sussman. *Structure and Interpretation of Computer Programs.* MIT Press, Cambridge, Massachusetts, 1985.

[Aïit-Kaci, 1991] H. Aïit-Kaci. *Warren's Abstract Machine.* Logic Programming. MIT Press, Cambridge, Massachusetts, 1991.

[Allen, 1978] J. Allen. *The Anatomy of Lisp.* McGraw-Hill, New York, 1978.

[Barendregt, 1984] H. Barendregt. *The Lambda Calculus: Its Syntax and Semantics.* North-Holland, Amsterdam, Revised Edition. Auflage, 1984.

[Bawden und Rees, 1988] A. Bawden und J. Rees. Syntactic Closures. In *Proc. of the 1988 ACM Conference on Lisp and Functional Programming*, Salt Lake City, Utah, Juli 1988. ACM Press.

[Bergin und Gibson, 1996] T. Bergin und R. Gibson (Hrsg.). *History of Programming Languages.* ACM Press, New York, 1996.

[Bishop, 1986] J. Bishop. *Data Abstraction in Programming Languages.* International Computer Science Series. Addison-Wesley, Wokingham, England, 1986.

[BMFT, 1994] BMFT. *Initiative zur Förderung der Software-Technologie in Wirtschaft, Wissenschaft und Technik.* Bundesministerium für Forschung und Technologie, Bonn, 1994.

[Bobrow *et al.*, 1986] D. G. Bobrow, K. Kahn, G. Kiczales, L. Masinter, M. Stefik und F. Zdybel. CommonLoops: Merging Lisp and Object-Oriented Programming. In *ACM OOPSLA '86*, Seite 17–29, 1986.

[Bobrow *et al.*, 1988] D. Bobrow, L. G. DeMichiel, R. P. Gabriel, S. E. Keene, G. Kiczales und D. A. Moon. Common Lisp Object System Specification. *Sigplan Notices*, 23(Special Issue), September 1988.

[Bobrow *et al.*, 1993] D. G. Bobrow, R. P. Gabriel und J. L. White. CLOS in Context: The Shape of the Design Space. In A. Paepcke (Hrsg.), *Object-Oriented Programming: The CLOS Perspective*, Seite 29–61. MIT Press, Cambridge, Massachusetts, 1993.

[Bobrow und Stefik, 1981] D. Bobrow und M. Stefik. The Loops Manual. Technical Report KB-VLSI-81-13, Xerox, Palo Alto, CA, 1981.

[Bobrow und Winograd, 1977] D. Bobrow und T. Winograd. An Overview of KRL, a Knowledge Representation Language. *Cognitive Science*, 1(1):3–46, 1977. Auch erschienen in [Brachman und Levesque, 1985].

[Boehm und Weiser, 1988] H. J. Boehm und M. Weiser. Garbage Collection in an Uncooperative Environment. *Software - Practice and Experience*, 18(9), September 1988.

[Boehm, 1981] B. W. Boehm. *Software Engineering Economics*. Prentice Hall, Englewood Cliffs, NJ, 1981.

[Boehm, 1988] B. W. Boehm. A Spiral Model of Software Development and Enhancement. *IEEE Computer*, Seite 61–72, Mai 1988.

[Booch, 1991] G. Booch. *Object Oriented Design with Applications*. Ada and Software Engineering. The Benjamin/Cummings Publishing Company, Inc., Redwood City, California, 1991.

[Booch, 1994] G. Booch. *Object Oriented Design with Applications*. Ada and Software Engineering. The Benjamin/Cummings Publishing Company, Inc., Redwood City, California, 1994.

[Bracha und Cook, 1990] G. Bracha und W. Cook. Mixin-based Inheritance. In *Proc. of the ECOOP/OOPSLA '90*, SIGPLAN Notices, Seite 303–311. ACM, Oktober 1990.

[Bracha und Lindstrom, 1992] G. Bracha und G. Lindstrom. Modularity meets Inheritance. In *Proc. IEEE Computer Society International Conference on Computer Languages*, Seite 282–290, Washington DC, April 1992. IEEE Computer Society.

[Bracha, 1992] G. Bracha. *The Programming Language Jigsaw: Mixins, Modularity and Multiple Inheritance*. Doktorarbeit, Departement of Computer Science, University of Utah, März 1992.

[Brachman und Levesque, 1985] R. J. Brachman und H. J. Levesque (Hrsg.). *Readings in Knowledge Representation*. Morgan Kaufmann Publishers, Los Altos, CA, 1985.

[Brachman und Schmolze, 1985] R. J. Brachman und J. G. Schmolze. An Overview of the KL-ONE Knowledge Representation System. *Cognitive Science*, 9:171–216, 1985.

[Bradford, 1996] R. J. Bradford. An Implementation of Telos in Common Lisp. *Object-Oriented Systems*, 3:31–49, 1996.

[Brandt, 1995] S. Brandt. Reflection in a Statically Typed and Object-Oriented Language - A Meta-Level Interface for BEA. Interner Bericht, Aarhus University, Aarhus, Juli 1995.

[Bretthauer *et al.*, 1989a] H. Bretthauer, T. Christaller und J. Kopp. Multiple vs. Single Inheritance in Object-oriented Programming Languages. What do we really want? In *Workshop der Fachgruppe 2.1.4. „Alternative Konzepte für Sprachen und Rechner" der Gesellschaft für Informatik*, Bad Honnef, 1989.

[Bretthauer *et al.*, 1989b] H. Bretthauer, T. Christaller und J. Kopp. Multiple vs. Single Inheritance in Object-oriented Programming Languages. *Microprocessing and Microprogramming*, 28:197–200, 1989.

[Bretthauer *et al.*, 1989c] H. Bretthauer, T. Christaller und J. Kopp. Multiple vs. Single Inheritance in Object-oriented Programming Languages. What do we really want? Arbeitspapiere der GMD 415, Gesellschaft für Mathematik und Datenverarbeitung (GMD), Sankt Augustin, November 1989.

[Bretthauer *et al.*, 1992] H. Bretthauer, H. Davis, J. Kopp und K. Playford. Balancing the EuLisp Metaobject Protocol. In A. Yonezawa und B. C. Smith (Hrsg.), *IMSA'92 Workshop on Reflection and Meta-Level Architecture*, Seite 113–118, Tokio, 1992.

[Bretthauer *et al.*, 1993] H. Bretthauer, J. Kopp, H. Davis und K. Playford. Balancing the EuLisp Metaobject Protocol. *Lisp and Symbolic Computation*, 6(1/2):119–138, August 1993.

[Bretthauer *et al.*, 1994] H. Bretthauer, T. Christaller, H. Friedrich, W. Goerigk, W. Heicking, U. Hoffmann, A. Kind, B. Klude, H. Knutzen, J. Kopp, U. Kriegel, I. Mohr, R. Rosenmüller und F. Simon. Von der APPLY-Methodik zum System. APPLY-Arbeitspapier APPLY/XIII/10, CAU/GMD/ISST/VW-GEDAS, Sankt Augustin, Juni 1994.

[Bretthauer und Kopp, 1991] H. Bretthauer und J. Kopp. The Meta-Class-System MCS. A Portable Object System for Common Lisp. Documentation. Arbeitspapiere der GMD 554, Gesellschaft für Mathematik und Datenverarbeitung (GMD), Sankt Augustin, Juli 1991.

[Broadbery und Burdorf, 1993] P. Broadbery und C. Burdorf. Applications of TELOS. *Lisp and Symbolic Computation*, 6(1/2):139–158, August 1993.

[Brooks, 1996] J. F. P. Brooks. Language Design as Design. In T. Bergin und R. Gibson (Hrsg.), *History of Programming Languages*, Seite 4–16. ACM Press, New York, 1996.

[Buschmann *et al.*, 1992] F. Buschmann, K. Kiefer, F. Paulisch und M. Stal. The Meta-Information-Protocol: Run-Time Type Information for C++. In A. Yonezawa und B. C. Smith (Hrsg.), *IMSA'92 Workshop on Reflection and Meta-Level Architecture*, Seite 82–87, Tokio, 1992.

[Carbonell, 1970] J. R. Carbonell. AI in CAI: An Artificial Intelligence Approach to Computer-Assisted Instruction. *IEEE Transactions on Man-Machine Systems*, 11(4):190–202, Dezember 1970.

[Cardelli und Wegner, 1985] L. Cardelli und P. Wegner. On Understanding Types, Data Abstraction and Polymorphism. *ACM Computing Surveys*, 17(4), 1985.

[Carre und Geib, 1990] B. Carre und J.-M. Geib. The Point of View notion for Multiple Inheritance. In *Proc. of the ECOOP/OOPSLA '90*, SIGPLAN Notices, Seite 312–321. ACM, Oktober 1990.

[Chailloux *et al.*, 1991] J. Chailloux, M. Devin, F. Dupont, J.-M. Hullot und J. Serpette, B. anad Vuillemin. *Le-Lisp version 15.24, le manual de reference.* INRIA, Rocquencourt, Mai 1991.

[Chambers *et al.*, 1989] C. Chambers, D. Ungar und E. Lee. An Efficient Implementation of SELF, a Dynamically-Typed Object-Oriented Language Based on Prototypes. In *Proc. of the OOPSLA '89*, Band 24 der *SIGPLAN Notices*, Seite 49–70, New Orleans, LA, Oktober 1989. ACM. Auch erschienen in [SELF, 1991].

[Christaller *et al.*, 1989] T. Christaller, F. di Primio und A. Voß (Hrsg.). *Die KI-Werkbank BABYLON.* Addison-Wesley, Bonn, 1989.

[Christaller, 1986] T. Christaller. Einführung in Lisp. Vorlesungsskript, KIFS 86, Sankt Augustin, 1986.

[Christaller, 1987] T. Christaller. *Die Entwicklung generischer Konstrollstrukturen aus kaskadierten ATNs.* Doktorarbeit, Univeristät Hamburg, Januar 1987.

[Christaller, 1988] T. Christaller. Eine Einführung in LISP. In T. Christaller, H.-W. Hein und M. M. Richter (Hrsg.), *Künstliche Intelligenz. Theoretische Grundlagen und Anwendungsfelder*, Band 159 der *Informatik-Fachberichte*, Seite 1–35. Springer-Verlag, Berlin, 1988.

[Christaller, 1997] T. Christaller. Warum es kein Museum für Software gibt aber geben sollte. August 1997.

[Clinger und Rees, 1991] W. Clinger und J. Rees. Macros that work. In *POPL '91 - Eighteens Annual ACM Symposium on Principles of Programming Languages*, Seite 155–162, Orlando, Florida, 1991. ACM Press.

[Coad und Yourdon, 1994] P. Coad und E. Yourdon. *Object Oriented Ananysis.* Prentice Hall, Englewood Cliffs, NJ, 1994.

[Cointe, 1987] P. Cointe. Metaclasses are First Class: the ObjVlisp Model. *SIGPLAN Notices*, 22(12), Dezember 1987.

[Cointe, 1988] P. Cointe. The ObjVlisp Kernel: a Reflective Lisp Architecture to define a Uniform Object-Oriented System. In P. Maes und D. Nardi (Hrsg.), *Meta-Level Architectures and Reflection*, Seite 155–176. North Holland, Amsterdam, Niederlande, 1988.

[Cointe, 1993] P. Cointe. CLOS and Smalltalk. In A. Paepcke (Hrsg.), *Object-Oriented Programming: The CLOS Perspective*, Seite 215–250. MIT Press, Cambridge, Massachusetts, 1993.

[Coplien, 1992] J. O. Coplien. *Advanced C++ Programming Styles and Idioms.* Addison-Wesley, 1992.

[Cox, 1991] B. J. Cox. *Object-Oriented Programming, An Evolutionary Approach.* Addison-Wesley, Reading, Massachusetts, 1991.

[Dahl und Nygaard, 1966] O.-J. Dahl und K. Nygaard. Simula – an ALGOL-Based Simulation Language. *Communications of the ACM*, 9(9):671–678, September 1966.

[de Buhr und Friederich, 1987] E. de Buhr und V. Friederich. Das WEREX-Flavor-System - eine Basis für objektorientierte Programmierung in Lisp-Umgebungen. WEREX-Bericht Nr. 2, ADV-ORGA, Wilhelmshafen, Mai 1987.

[DeMichiel, 1993a] L. G. DeMichiel. An Introduction to CLOS. In A. Paepcke (Hrsg.), *Object-Oriented Programming: The CLOS Perspective*, Seite 3–27. MIT Press, Cambridge, Massachusetts, 1993.

[DeMichiel, 1993b] L. G. DeMichiel. CLOS and C++. In A. Paepcke (Hrsg.), *Object-Oriented Programming: The CLOS Perspective*, Seite 157–180. MIT Press, Cambridge, Massachusetts, 1993.

[Deutsch, 1989] P. L. Deutsch. The Past, Present, and Future of Smalltalk. In S. Cook (Hrsg.), *ECOOP'89*, Seite 73–87, Nottingham, Juli 1989. Cambridge University Press.

[di Primio und Christaller, 1983] F. di Primio und T. Christaller. A Poor Man's Flavor System. Interner Bericht, ISSCO, Universität Genf, Genf, 1983.

[di Primio, 1988] F. di Primio. Micro Common Flavors. Arbeitspapiere der GMD 295, GMD, Sankt Augustin, Februar 1988.

[di Primio, 1993] F. di Primio. *Hybride Wissensrepräsentation – Am Beispiel von BABYLON*. Deutscher Universitäts-Verlag, Wiesbaden, 1993.

[Drescher, 1987] G. Drescher. *ObjectLISP User Manual*. Lisp Machine Inc., Cambridge, Massachusetts, 1987.

[Ducournau *et al.*, 1992] R. Ducournau, M. Habib, M. Huchard und M. Mugnier. Monotonic Conflict Resolution Mechanisms for Inheritance. In *OOPSLA '92 - Object-Oriented Programming Systems and Languages*, Seite 4–16. ACM, 1992.

[Ducournau und Habib, 1991] R. Ducournau und M. Habib. Masking and Conflicts, or To Inherit is Not To Own! In M. Lenzerini, D. Nardi und M. Simi (Hrsg.), *Inheritance Hierarchies in Knowledge Representation and Programming Languages*, Kapitel 14, Seite 223–244. John Wiley, 1991.

[Ellis und Stroustrup, 1990] M. A. Ellis und B. Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, Reading, Massachusetts, 1990.

[Emde *et al.*, 1996] W. Emde, C. Beilken, J. Börding, U. Petersen, M. Spenke, A. Voss und S. Wrobel. Configuration of Telecommunication Systems in KIKon. In B. Faltings und E. Freuder (Hrsg.), *Configuration - Papers from the 1996 Fall Symposium*, Seite 105–110, Menlo Park: AAAI Press, 1996.

[Fichman und Kemerer, 1992] R. G. Fichman und C. F. Kemerer. Object-Oriented and Conventional Analysis and Design Methods - Comparison and Critique. *IEEE Computer*, 25(10):22–40, Oktober 1992.

[Fischer, 1994] G. Fischer. *Programming Languages - Principles and Praxis.* Ada and Software Engineering. The Benjamin/Cummings Publishing Company, Inc., Redwood City, California, 1994.

[Flatt *et al.*, 1998] M. Flatt, S. Krishnamurthi und M. Felleisen. Classes and Mixins. In *Proceedings 25th Annual ACM Symposium on Principles of Programming Languages*, San Diego, CA, Januar 1998.

[Friedman *et al.*, 1992] D. P. Friedman, M. Wand und C. Haynes. *Essentials of Programming Languages.* MIT Press, Cambridge, Massachusetts, 1992.

[Gabriel, 1993] R. P. Gabriel. LISP: good news, bad news, how to win BIG. *AI Expert*, Seite 31–39, Juni 1993.

[Goerigk, 1993] W. Goerigk. *Korrektheit der Übersetzung objektorientierter Wissensrepräsentationssprachen mit statischer Vererbung.* Doktorarbeit, Christian-Albrechts-Universität Kiel, Kiel, Februar 1993. 9304.

[Goerigk, 1997] W. Goerigk. Towards Rigorous Compiler Implementation Verification. In *Workshop der Fachgruppe 2.1.4. "Alternative Konzepte für Sprachen und Rechner" der Gesellschaft für Informatik*, Bad Honnef, April 1997. GI.

[Goldberg und Robson, 1983] A. Goldberg und D. Robson. *Smalltalk-80: The Language and its Implementation.* Computer Science. Addison-Wesley, Reading, Massachusetts, 1983.

[Goldberg und Robson, 1989] A. Goldberg und D. Robson. *Smalltalk-80. The Language.* Addison-Wesley, Reading, Massachusetts, 1989.

[Goldberg, 1984] A. Goldberg. The Influence of an Object-Oriented Language on the Programming Environment. In D. R. Barstow, H. E. Shrobe und E. Sandewall (Hrsg.), *Interactive Programming Environments*, Seite 141–174. McGraw-Hill, New York, 1984.

[Gosling *et al.*, 1996] J. Gosling, B. Joy und G. Steele. *The Java Language Specification.* The Java Series. Addison-Wesley, Reading, Massachusetts, 1996.

[Hewitt *et al.*, 1973] C. Hewitt, P. Bishop und R. Steiger. A Universal Modular ACTOR Formalism for Artificial Intelligence. In *Proc. of the 3th International Joint Conference on Artificial Intelligence*, Seite 235–245, Standford, CA, August 1973.

[Hommes *et al.*, 1980] F. Hommes, W. Kluge und H. Schlütter. A Reduction Machine Architecture and Expression-Oriented Editing. Internal report 80, 0004, Gesellschaft für Mathematik und Datenverarbeitung, Sankt Augustin, 1980.

[Horowitz, 1983] E. Horowitz. *Fundamentals of Programming Languages.* Springer, Berlin, 1983.

[Hudak und Wadler, 1992] P. Hudak und P. Wadler. Report on the Functional Programming Language Haskell. *SIGPLAN Notices*, 27(7), Mai 1992.

[Hudak, 1989] P. Hudak. Conception, Evolution, and Application of Functional Programming Languages. *ACM Computing Surveys*, 21(3):359–411, September 1989.

[Hughes, 1990] J. Hughes. Why Functional Programming Matters. In D. A. Turner (Hrsg.), *Research Topics in Functional Programming*, University of Texas at Austin of Programming, Seite 17–42. Addison-Wesley, Reading, Massachusetts, 1990.

[IEEE Std 1178-1990, 1991] *IEEE Standard for the Scheme Programming Language*. Institute of Electrical and Electronic Engineers, New York, NY, 1991.

[IlogTalk, 1994] ILOG, SA, Gentilly, France. *IlogTalk Reference Manual*, 1994.

[Jacobson, 1994] I. Jacobson. Toward Mature - Object Technology. *ROAD*, 1(1), Mai 1994.

[Jalote, 1997] P. Jalote. *An Integrated Approach to Software Engineering*. Undergraduate Texts in Computer Science. Springer-Verlag New York Inc., New York, NY, 1997.

[Karbach, 1994] W. Karbach. *MODEL-K: Modellierung und Operationalisierung von Selbsteinschätzung und -steuerung durch Reflexion und Metawissen*. Reihe DISKI. infix Verlag, Sankt Augustin, 1994.

[Kay, 1996] A. Kay. The Early History of Smalltalk. In T. Bergin und R. Gibson (Hrsg.), *History of Programming Languages*, Seite 511–589. ACM Press, New York, 1996.

[Keene, 1989] S. E. Keene. *Object-Oriented Programming in Common Lisp. A Programmer's Guide to CLOS*. Addison-Wesley, Reading, Massachusetts, 1989.

[Kempf und Stelzner, 1987] R. Kempf und M. Stelzner. Teaching Object-Oriented Programming with the KEE System. *SIGPLAN Notices*, 22(12):11–25, Dezember 1987.

[Kiczales *et al.*, 1991] G. Kiczales, J. des Rivieres und D. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, Cambridge, Massachusetts, 1991.

[Kiczales und Rodriguez, 1990] G. Kiczales und L. Rodriguez. Efficient Method Dispatch in PCL. In *Proc. of the 1990 ACM Conference on Lisp and Functional Programming*, Seite 99–105, New York, 1990. ACM Press.

[Kiczales, 1996] G. Kiczales (Hrsg.). *International Conference on Reflection and Meta-Level Architecture*. San Francisco, 1996.

[Kind und Friedrich, 1993] A. Kind und H. Friedrich. A Practical Approach to Type Inference for EuLisp. *Lisp and Symbolic Computation*, 6(1/2):159–175, August 1993.

[Kind, 1998] A. Kind. *An Architecture for Interpreted Dynamic Object-Oriented Languages*. Doktorarbeit, University of Bath, Bath, UK, Mai 1998.

[Kluge, 1997] W. Kluge. *International Workshop on the Implementation of Functional Languages IFL '96*. Lecture notes in computer science. Springer, New York, August 1997.

[Kohlbecker *et al.*, 1986] E. E. Kohlbecker, D. P. Friedman, M. Felleisen und B. Duba. Hygienic Macro Expansion. In *Proc. of the 1986 ACM Symposium on Lisp and Functional Programming*, Seite 151–161, New York, August 1986. ACM Press.

[Kopp, 1996a] J. Kopp. *Konstruktion von Wissensrepräsentationssprachen durch Nutzen und Erweitern objektorientierter Sprachmittel.* Reihe DISKI. infix Verlag, Sankt Augustin, 1996.

[Kopp, 1996b] J. Kopp. Persönliche Kommunikation. 1996.

[Kriegel, 1992] E. U. Kriegel. Memory Management in APPLY. APPLY-Arbeitspapier APPLY/ISST/VI.1/3, Fraunhofer Institut für Software und Systemtechnik (ISST), Berlin, April 1992.

[Kristensen, 1996] B. B. Kristensen. A Conceptual Perspective on the Comparison of Object-Oriented Programming Languages. *ACM SIGPLAN Notices*, 31(2):42–54, Februar 1996.

[Laddaga und Veitch, 1997] R. Laddaga und J. Veitch. Dynamic Object Technology. *Communications of the ACM*, 40(5), Mai 1997.

[Lang und Pearlmutter, 1986] K. J. Lang und B. A. Pearlmutter. Oaklisp: An Object-Oriented Scheme with First Class Types. In *ACM Conference on Object-Oriented Systems, Programming, Languages and Applications*, Seite 30–37. ACM, September 1986.

[Lang und Pearlmutter, 1988] K. J. Lang und B. A. Pearlmutter. Oaklisp: An Object-Oriented dialect of Scheme with First Class Types. *Lisp and Symbolic Computation*, 1(1):39–51, Mai 1988.

[Lange, 1993] A. Lange. Eine CLOS-kompatible objektorientierte Erweiterung für Scheme. Diplomarbeit, Universität Bonn, Bonn, Februar 1993.

[Laubsch, 1982] J. Laubsch. ObjTalk: Eine Lisp-Erweiterung zum objekt-orientierten Programmieren. MMK-Memo 22, Universität Stuttgart, Januar 1982.

[LeLisp, 1992] ILOG, SA, Gentilly, France. *Le-Lisp version 16 Reference Manual*, 1992.

[Lieberman, 1981] H. Lieberman. A Preview of Act 1. AI Memo 625, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, Cambridge, Massachusetts, Juni 1981.

[Liskov, 1974] S. Liskov, Barbara an Zilles. Programming with Abstract Data Types. *SIGPLAN Notices*, 9(4):50–59, Juni 1974.

[Lorenz, 1995] M. Lorenz. *Rapid Software Development with Smalltalk.* Advances in Object Technology. SIGS Books, New York, 1995.

[Louden, 1994] X. Louden. *Programming Languages - Principles and Praxis.* Ada and Software Engineering. The Benjamin/Cummings Publishing Company, Inc., Redwood City, California, 1994.

[Madsen *et al.*, 1993] O. L. Madsen, B. Moeller-Pedersen und K. Nygaard. *Object-Oriented Programming in the BETA Programming Language.* ACM Press Books. Addison-Wesley, Wokingham, England, 1993.

[Maes und Nardi, 1988] P. Maes und D. Nardi (Hrsg.). *Meta-Level Architectures and Reflection*. North Holland, Amsterdam, Niederlande, 1988.

[Maes, 1987] P. Maes. Computational Reflection. Technical report 87-2, Vrije Universiteit Brüssel, AI-Laboratory, 1987.

[Martin und Odell, 1992] J. Martin und J. J. Odell. *Object-Oriented Analysis and Design*. Prentice-Hall, Englewood Cliffs, NJ, 1992.

[McCarthy, 1981] J. McCarthy. The Evolution of Lisp. In R. L. Wexelblat (Hrsg.), *History of Programming Languages*, ACM Monograph, Seite 173–197. Academic Press, New York, 1981.

[Meyer, 1988] B. Meyer. *Object-oriented Software Construction*. Prentice Hall, New York, 1988.

[Meyer, 1992] B. Meyer. *Eiffel: The Language*. Prentice Hall Object-Oriented Series. Prentice Hall, New York, 1992.

[Milner *et al.*, 1990] R. Milner, M. Tofte und R. Harper. The Definition of Standard ML. Interner Bericht, MIT Press, Cambridge, Massachusetts, 1990.

[Minsky, 1975] M. Minsky. A Framework for Representing Knowledge. In P. Winston (Hrsg.), *The Psychology of Computer Vision*, Seite 211–277, New York, 1975. McGraw-Hill.

[Moon, 1984] D. A. Moon. Garbage Collection in Large Lisp Systems. In *Proc. of the 1984 ACM Conference on Lisp and Functional Programming*, Seite 235–246, New York, 1984. ACM Press.

[Moon, 1986] D. A. Moon. Object-Oriented Programming with Flavors. In *Proc. of the OOPSLA '86*, Seite 1–8, Portland, Oregon, September 1986.

[Müller, 1986] B. S. Müller. Lehrmaterialien BABYLON: Die Beispielswissensbasen zur Pilzbestimmung. Arbeitspapiere der GMD 221, Gesellschaft für Mathematik und Datenverarbeitung, Sankt Augustin, 1986.

[Nagl, 1982] M. Nagl. *Einführung in die Programmiersprache Ada*. uni-text. Vieweg, Braunschweig, 1982.

[Newell und Simon, 1972] A. Newell und H. Simon. Computer Science as Empirical Inquiry: Symbols and Search. *Communications of ACM*, 19(2):42–54, Februar 1972.

[Padget *et al.*, 1986] J. Padget, J. Chailloux, T. Christaller, R. deMantaras, J. Dalton, M. Devin, J. F. Fitch, T. Krummnack, E. Neidl, E. Papon, S. Pope, C. Queinnec, L. Steels und H. Stoyan. Desiderata for the Standardisation of Lisp. In *Proc. of the 1986 ACM Symposium on Lisp and Functional Programming*, Seite 54–66, New York, 1986. ACM Press.

[Padget *et al.*, 1993] J. Padget, G. Nuyens und H. Bretthauer. An Overview of EuLisp. *Lisp and Symbolic Computation*, 6(1/2):9–98, August 1993.

[Paepcke, 1988] A. Paepcke. PCLOS: A Flexible Implementation of CLOS Persistence. In S. Gjessing und K. Nygaard (Hrsg.), *Proc. of the ECOOP '88*, Lecture Notes in Computer Science, Seite 157–180. Springer Verlag, 1988.

[Paepcke, 1990] A. Paepcke. PCLOS: Stress Testing CLOS. In *Proc. of the ECOOP/OOPSLA '90*, SIGPLAN Notices, Seite 194–211, Ottawa, Canada, Oktober 1990. ACM.

[Paepcke, 1993] A. Paepcke (Hrsg.). *Object-Oriented Programming: The CLOS Perspective*. MIT Press, Cambridge, Massachusetts, 1993.

[Paulson, 1991] L. P. Paulson. *ML for the Working Programmer*. Cambridge University Press, Cambridge, 1991.

[Poeppel, 1985] E. Poeppel. *Grenzen des Bewußtseins*. Deutsche Verlags-Anstalt, Stuttgart, 1985.

[Queinnec, 1993] C. Queinnec. Designing MEROON v3. In C. Rathke, J. Kopp, H. Hohl und H. Bretthauer (Hrsg.), *Object-Oriented Programming in Lisp: Languages and Applications. A Report on the ECOOP '93 Workshop*, Band 788 der *Arbeitspapiere der GMD*, Seite 19–32, Sankt Augustin, Germany, September 1993. GMD.

[Queinnec, 1995] C. Queinnec. Dmeroon: Overview of a Distributed Class-Based Causally-Coherent Data Model. In T. Ito, R. H. J. Halstead und C. Queinnec (Hrsg.), *PSLS 95 - Parallel Symbolic Langages and Systems*, Band 1068 der *Lecture Notes in Computer Science*, Seite 297–309, Beaune, France, Oktober 1995.

[Queinnec, 1996] C. Queinnec (Hrsg.). *Lisp in Small Pieces*. Cambridge University Press, Cambridge, England, 1996.

[Queinnec, 1997] C. Queinnec. Fast and Compact Dispatching for Dynamic Object-Oriented Languages. *Information Processing Letters*, 1997.

[Quillian, 1967] M. Quillian. Word Concepts: A Theory and Simulation of Some Basic Semantic Capabilities. *Behavioral Science*, 12:410–430, 1967. Auch erschienen in [Brachman und Levesque, 1985].

[Rathke und Laubsch, 1983] C. Rathke und J. Laubsch. OBJTALK:. In *Objektorientierte Software- und Hardwarearchitekturen*. B. G. Teubner, Stutgart, 1983.

[Roberts und Goldstein, 1977] R. B. Roberts und I. P. Goldstein. The FRL Primer. AI Memo 408, MIT, Cambridge, Massachusetts, Juli 1977.

[Rose, 1988] J. R. Rose. Fast Dispatch Mechanisms for Stock Hardware. In *Proceedings of the OOPSLA '88*, SIGPLAN Notices, Seite 27–35, New Orleans, LA, September 1988. ACM.

[Rose, 1991] J. R. Rose. A Minimal Metaobject Protocol for Dynamic Dispatch. In *Proceedings of the OOPSLA '91 Workshop on Reflection and Metalevel Architecture in Object-Oriented Programming*, Oktober 1991.

[Rumbaugh, 1991] J. e. a. Rumbaugh. *Object-Oriented Modelling and Design.* Prentice Hall, 1991.

[Sandewall, 1984] E. Sandewall. Programming in an Interactive Environment: The Lisp Experience. In D. R. Barstow, H. E. Shrobe und E. Sandewall (Hrsg.), *Interactive Programming Environments*, Seite 31–80. McGraw-Hill, New York, 1984.

[Schank und Riesbeck, 1981] R. C. Schank und C. K. Riesbeck. *Inside Computer Understanding: Five Programs Plus Miniatures.* Artificial Intelligence. Lawrence Erlbaum Associates, Hillsdale, N.J., 1981.

[Schmidt und Omohundro, 1993] H. W. Schmidt und S. M. Omohundro. CLOS, Eiffel, and Sather. In A. Paepcke (Hrsg.), *Object-Oriented Programming: The CLOS Perspective*, Seite 181–213. MIT Press, Cambridge, Massachusetts, 1993.

[SELF, 1991] Special Issue on SELF. *Lisp and Symbolic Computation*, 4(3), Juli 1991.

[Shalit, 1996] A. Shalit. *The Dylan Reference Manual.* Apple Press. Addison-Wesley, Reading, Massachusetts, 1996.

[Shl,]

[Shriver und Wegner, 1987] B. Shriver und P. Wegner (Hrsg.). *Research Directions in Object-Oriented Programming.* Computer Systems Series. MIT Press, Cambridge, Massachusetts, 1987.

[Smith und Ungar, 1995] R. B. Smith und D. Ungar. Programming as an Experience: The Inspiration for Self. In *ECOOP '95 Conference Proceedings*, 1995.

[Smith, 1982] B. C. Smith. *Reflection and Semantics in a Procedural Language.* Doktorarbeit, MIT Laboratory for Computer Science, Cambridge, Massachusetts, September 1982. auch als MIT technical report 272 publiziert.

[Smith, 1984] B. C. Smith. Reflection and Semantics in LISP. In *Proceedings 11th Annual ACM Symposium on Principles of Programming Languages*, Seite 23–35, Salt Lake City, Utah, Januar 1984.

[Smolka, 1995] G. Smolka. An Oz Primer. DFKI Oz Documentation Series, DFKI, April 1995.

[Springer und Friedman, 1989] G. Springer und D. P. Friedman. *Scheme and the Art of Programming.* MIT Press, Cambridge, Massachusetts, 1989.

[Steele Jr., 1984] G. L. Steele Jr. *Common Lisp - The Language.* Digital Press, Bedford, Massachusetts, 1984.

[Steele Jr., 1990a] G. L. Steele Jr. *Common Lisp - The Language, Second Edition.* Digital Press, Bedford, Massachusetts, 1990.

[Steele Jr., 1990b] G. L. Steele Jr. *Common Lisp - The Language, Second Edition.* Digital Press, Bedford, Massachusetts, 1990.

[Steele und Gabriel, 1996] G. L. Steele und R. P. Gabriel. The Evolution of Lisp. In T. Bergin und R. Gibson (Hrsg.), *History of Programming Languages*, Seite 233–330. ACM Press, New York, 1996.

[Stoyan und Görz, 1983] H. Stoyan und G. Görz. Was ist objektorientierte Programmierung? In *Objektorientierte Software- und Hardwarearchitekturen*. B.G. Teubner, Stuttgart, 1983.

[Stoyan und Görz, 1984] H. Stoyan und G. Görz. *LISP - Eine Einführung in die Programmierung*. Studienreihe Informatik. Springer-Verlag, Berlin, 1984.

[Stoyan, 1980] H. Stoyan. *LISP - Anwendungsgebiete, Grundbegriffe, Geschichte*. Akademie-Verlag, Berlin, 1980.

[Strachey, 1967] C. Strachey. Fundamental Concepts in Programming Languages. Interner Bericht, Lecture Notes for International Summer School in Computer Programming, Kopenhagen, August 1967.

[Strousrup, 1996] B. Strousrup. The Design and Evolution of C++. In T. Bergin und R. Gibson (Hrsg.), *History of Programming Languages*, Seite 699–764. ACM Press, New York, 1996.

[Stroustrup, 1987] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, Massachusetts, 1987.

[Stroustrup, 1988] B. Stroustrup. What is Object-Oriented Programming? *IEEE Software*, Seite 10–20, Mai 1988.

[Stroustrup, 1991] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, Massachusetts, 2. Auflage, 1991.

[Stroustrup, 1994] B. Stroustrup. *The Design and Evolution of C++*. Addison-Wesley, Reading, Massachusetts, 1994.

[Symbolics Inc, 1985] Symbolics Inc., Cambridge, Massachusetts. *Refererence Guide to Symbolics-Lisp*, 5. Auflage, 1985.

[Szyperski, 1992] C. A. Szyperski. Import is Not Inheritance – Why We Need Both: Modules and Classes. In O. L. Madsen (Hrsg.), *Proc. of the European Conference on Object-Oriented Programming*, Band 615 der *Lecture Notes in Computer Science*, Seite 19–32, Utrecht, Juli 1992. Springer-Verlag.

[Templ, 1994] J. Templ. Metaprogramming in Oberon. Interner Bericht, ETH Zürich, Dissertation, Zürich, Juli 1994.

[Touretzky, 1986] D. S. Touretzky. *The Mathematics of Inheritance Systems*. Morgan Kaufmann Publishers, Los Altos, CA, 1986.

[Turner, 1990] D. A. Turner. An Overview of Miranda. In D. A. Turner (Hrsg.), *Research Topics in Functional Programming*, University of Texas at Austin of Programming, Seite 1–16. Addison-Wesley, Reading, Massachusetts, 1990.

[Ungar, 1988] D. Ungar. Generation scavening: A Non-Disruptive High Performance Storage Reclamation Algorithm. *SIGPLAN Notices*, 19(5):157–167, 1988.

[Vitek *et al.*, 1997] J. Vitek, R. N. Horspool und A. Krall. Efficient Type Inclusion Tests. In *Proc. of the OOPSLA '97*, SIGPLAN Notices, Seite 142–157. ACM, Oktober 1997.

[Vitek und Horspool, 1994] J. Vitek und R. N. Horspool. Taming Message Passing: Efficient Method Look-Up for Dynamically-Typed Languages. In *Proc. of the ECOOP '94*, Lecture Notes in Computer Science. Springer Verlag, 1994.

[Voß, 1995] A. Voß. C++, Common Lisp/CLOS, Eiffel oder Smalltalk? *KI*, 2:35–44, März 1995.

[VW-Gedas, 1991] VW-GEDAS GmbH, Berlin. *babylon Version 3.0: Softwarewerkzeuge für industrielle Anwendungen. Referenzhandbuch*, 1991.

[Wahlster, 1997] W. Wahlster. VERBMOBIL: Erkennung, Analyse, Transfer, Generierung und Synthese von Spontansprache. Verbmobil-Report 198, DFKI GmbH, Saarbrücken, 1997.

[Walther *et al.*, 1989] J. Walther, H. Bretthauer und J. Kopp. Portierungsanleitung für Babylon. In T. Christaller, F. di Primio und A. Voß (Hrsg.), *Die KI-Werkbank BABYLON*, Reihe Künstliche Intelligenz, Seite 403–412. Addison-Wesley, Bonn, 1989.

[Walther, 1988] J. Walther. Messungen zur Performanz von Objekt- und Lispsystemen. WEREX-Bericht Nr. 30, GMD, Sankt Augustin, Dezember 1988.

[Warren, 1983] D. H. D. Warren. An Abstract Prolog Instruction Set. Technical Note 309, SRI International, Menlo Park, CA, Oktober 1983.

[Wegner, 1987] P. Wegner. The Object-Oriented Classification Paradigm. In B. Shriver und P. Wegner (Hrsg.), *Research Directions in Object-Oriented Programming*, Seite 479–560. The MIT-Press, Cambridge u.a., 1987.

[Weinreb und Moon, 1981] Weinreb und Moon. *Lisp Machine Manual.* MIT, AI Laboratory, Cambridge, Massachusetts, 1981.

[Weizenbaum, 1968] J. Weizenbaum. The FUNARG Problem Explained. unpublished memorandum, MIT, Cambridge, Massachusetts, 1968.

[Winston, 1975] P. H. Winston. Learning Structural Descriptions from Examples. In P. Winston (Hrsg.), *The Psychology of Computer Vision*, Seite 157–209. McGraw-Hill, New York, 1975. Auch erschienen in [Brachman und Levesque, 1985].

[Wirth, 1985a] N. Wirth. *Programming in Modula-2.* Text and Monographs in Computer Science. Springer-Verlag, Berlin, 3. Edition. Auflage, 1985.

[Wirth, 1985b] N. Wirth. *Programming in Oberon.* Text and Monographs in Computer Science. Springer-Verlag, Berlin, 1985.

[Wirth, 1994] N. Wirth. Gedanken zur Software-Explosion. *Informatik Spektrum*, 17(1):5–10, Februar 1994.

[Yonezawa und Smith, 1992] A. Yonezawa und B. C. Smith (Hrsg.). *IMSA'92 Workshop on Reflection and Meta-Level Architecture*. Tokio, 1992.

[Zendra *et al.*, 1997] O. Zendra, D. Colnet und C. Suzanne. Efficient Dynamic Dispatch without Virtual Function Tables. The SmallEiffel Compiler. In *Proc. of the OOPSLA '97*, SIGPLAN Notices, Seite 125–141. ACM, Oktober 1997.

[Zimmer, 1991] R. M. Zimmer. Zur Pragmatik eines operationalisierten Lambda-Kalküls als Basis für interaktive Reduktionssysteme. Dissertation, GMD-Bericht, 0192, GMD, 1991.

Index

Abschlußeigenschaften, 277
Abstraktion, 26
    algorithmische, 26
    Daten-, 26
ACTOR, 26
Ada, 26
After-Methode, 38
Aktor-Sprachen
    Act1, 34
    ACTOR, 34
allgemeiner Konstruktor, 83
allocate, 84
Allokation
    von Objekten, 83
Allokationsoperationen
    abstrakte, 108
    primitive, 108
Allokationsprotokoll, 107, 243
Analyse
    Datenfluß-, 123
    Kontrollfluß-, 123
ancestor, 34
anwendbare Methode, 82
Anwendung
    komplexe, 13
Applet, 46
Applikationskompilation, 121
APPLY, 73
apply, 111
Around-Methode, 38
Attribut, 76, 287
Ausnahmebehandlung, 75

BABYLON, 9, 151, 152, 183
babylon, 10
Backquote-Ausdruck, 126
base class, 34
Basisklassen, 94
Before-Methode, 38
BETA, 38
Bezeichner
    Gültigkeit von, 73
    Sichtbarkeit von, 73
Bootstrapping, 118, 131, 173
Bytecode
    -Compiler, 118
    -Interpretierer, 118

C++, 33, 34, 38, 39, 42, 68
Cache
    Dispatch-, 141
Caching, 141
call-next-method, 38, 82
callNextMethod, 82, 89, 111, 128
CELOS, 222, 245
change-class, 264
child, 34
class-direkt-superklasses, 103
Classifier, 264
CLICC, 222
CLOS, 39, 53, 68, 194, 277
CLOS MOP, 38, 63
Closures, 104, 112, 129
Cobol, 26
Common Lisp, 48, 53
compile, 130
Compiler
    Applikations-, 123
    Bytecode-, 118
    inkrementeller, 118, 130
    Komplett-, 123
    Modul-, 121
Compilerbau, 26
compute-class-precedence-list, 103

Datei-Kompilation, 120
Datenabstraktion, 26
Datenflußanalyse, 123

Datentypen, 26
defclass, 228, 240
defgeneric, 232, 240
define class, 113
define function, 113
define generic, 113
defmethod, 232, 234, 240
Delegieren, 30, 31, 35
derived class, 34
descendant, 34
Designkriterien, 16, 23, 223, 278, 279
    Abstraktion, 16
    Effizienz, 22
    Einfachheit, 20
    Erweiterbarkeit, 19
    Generalisieren, 17
    Hierarchiebildung, 18
    Klassifikation, 17
    Komposition, 18
    Korrektheit, 21
    Modularisierung, 18
    Reflektion, 19
    Robustheit, 21
    sichere Abstraktion, 16
    Spezialisieren, 17
    Verursacherprinzip, 23
diskriminieren, 80
Dispatch
    -Cache, 141
    -Funktion, 143
    generischer, 82
    Methoden-, 82, 88
Dispatch-Cache, 141
Dispatch-Funktion, 143
DYLAN, 39, 48
dynamic binding, 38
dynamische Speicherverwaltung, 72
dynamische Struktur, 75
dynamisches Verhalten, 75

Effizienz, 22
Eiffel, 33, 34
Einbettungsmodell, 119
Einbettungstechnik, 126
Encapsulation, 32
Entwicklungssystem, 123
Eql-Methoden, 76, 82
Eql-Specializer, 59
Ersatzwert, 78
Erweiterbarkeit, 19
    syntaktische, 126
Erweiterungstechnik, 126
EU2C, 222
EULISP, 48, 219, 221
    level-0, 224
    level-1, 224
eval, 126, 130, 134, 145
exception handling, 75

Feld, 76, 77, 288
    Typannotation, 92
Feldannotationen, 77
    Allokationsart, 79
    Berechnung von, 103
    Ersatzwert, 78
    Initialisierungs-Schlüsselwort, 78
Feldzugriff, 159, 236
Feldzugriffsoperationen
    Berechnung von, 104, 106
Feldzugriffsprotokoll, 104, 240
Flavor, 93
FLAVORS, 49, 51, 53, 94, 151, 152
Fortran, 26
Frames, 26
freie Klasse, 94
funargs, 130
funcallable instances, 128
Funktion
    Dispatch-, 143
funktionale Objekte, 71
Funktionskompilation, 120

Gültigkeit, 73
garbage collection, 72
GC, 72
Generalisieren, 34
    von Objektklassen, 85
    von Struktur, 87, 91
    von Verhalten, 88, 92
generic lambda, 110
Generische Funktion, 56
generische Operation, 80, 137, 276
generischer Dispatch, 66, 82, 110, 139, 235, 245

einfach, 140
mehrfach, 141

höhere
Programmiersprachen, 26
Hierarchie
Instanziierungs-, 288
Klassen-, 18
Objekt-, 18

IlogTalk, 48
Implementierungstechnik, 8
Information-Hiding, 32
Initialisierung
generischer Operationen, 109
von Feldern, 100
von Klassen, 100
von Methoden, 109
von Objekten, 83
Initialisierungs-Schlüsselwort, 78, 103
Initialisierungsargument, 78
gefordert, 79
optional, 79
Initialisierungsprotokoll, 65, 107, 243
initialize, 84
Inline-Kompilierung, 123
inner, 38
Instanz, 31, 288
-Erzeugung, 159
-Initialisierung, 159
direkte, 287
indirekte, 288
terminale, 288
Instanziierung, 288
Instanziierungsgraph, 288
Instanziierungshierarchie, 288
Instanzvariable, 76
Interpretation, 117
Quellcode-, 145
Interpretierer, 118, 130
Bytecode-, 118
Introspektionsprotokoll, 65, 98, 238

JAVA, 33, 39, 45, 68, 271
JIT, 124
Just-In-Time-Compiler, 124

Kapselung, 32, 73
KIKon, 12
KIKONL, 12
Klasse, 31, 272, 288
abstrakte, 287
Basis-, 94
differenzieren, 287
freie, 94
Meta-, 289
Metaobjekt-, 97
Mixin-, 94, 289
Primär-, 94
Klasse-Instanz, 54
Klassenannotation, 84
constructor, 84
Klassendefinition, 75, 156, 228, 240
Klassenhierarchie, 64, 86, 97, 226, 236, 289
Klasseninitialisierung, 100, 102
Klassenpräzedenzliste, 76, 83, 86, 90
global konsistent, 86, 90
monoton, 87, 90
Klassenvariable, 289
Klassifizieren, 75
von Struktur, 75
von Verhalten, 75
KL-ONE, 12
kognitives Verarbeitungsmodell, 27
Kommunikation, 72
Kommunikationsmodell, 29, 35
Kompilation, 117
Applikations-, 121, 149
Datei-, 120
Funktions-, 120
inkrementelle, 120, 149
Komplett-, 8, 120, 121
Modul-, 8, 120, 121, 149
Komplettkompilation, 121
kongruente Signaturen, 81
Konstruktor
make, 83
spezieller, 83
Kontrollflußanalyse, 123
Korrektheit, 21

Lambda-Kalkül, 48
Laufzeitsystem, 123
Le-Lisp, 48

Lese- und Schreiboperationen, 59
Lisp, 26, 47

make, 107
Makroexpansion
    hygienisch, 127
    nicht-hygienisch, 127
Makros, 126, 222
    Backquote-Ausdruck, 126
    Einlese-, 126
    Read-, 126
Maschinensprache, 25
MCF, 49, 151
MCS, 151, 193
    MCS 0.5, 151, 277
    MCS 1.0, 193, 277
MEROONET, 49
message passing, 72, 289
Metaklasse, 289
metalevel architecture, 63
Metaobjekt, 63
    erzeugen, 238
    initialisieren, 238
Metaobjektebene, 63, 161, 224, 226, 236, 276
    syntaktische Erweiterung, 240
Metaobjektklassen, 97, 161
    Class, 97
    Field, 97
    GenericFunction, 97
    MetaObject, 97
    Method, 97
    SimpleClass, 97
    SimpleField, 97
    SimpleGenericFunction, 97
    SimpleMethod, 97
Metaobjektprotokoll, 38, 63, 289
method lambda, 110
Methode, 56, 80, 276, 289
    After-, 38
    anwendbare, 82
    Around-, 38
    Before-, 38
    Einfach-, 80
    Multi-, 80
    Primär-, 38
    Singleton-, 82
Methoden, 36, 75
    -Dispatch, 82
Methodenauswahl, 66, 110, 235, 245
Methodendefinition, 157, 232, 234, 240
Methodenkombination, 37, 38, 57, 59, 81, 82, 110, 157, 265
    CLOS, 265
    explizit, 265
    implizit, 269
    inner, 265
    super, 265
    TELOS, 268, 269
MFS, 39, 49, 151
Micro Flavor System, 49
middleware, 8
Mixin, 35, 93
Mixin-Klassen, 35, 94
Mixin-Vererbung, 203, 257
Modul, 19, 73, 221, 272
Modul-Compiler, 121
Modul-Kompilation, 121
Modula, 32
Modula-2, 26
Modularisierung, 18, 32, 224, 226
Modulkkonzept, 73
Monotonie, 87
MOP, 97
    addMethod, 110
    allocate, 107
    applyMethod, 111
    callMethod, 111
    classInstanceSize, 109
    classOf, 98
    compatibleSuperclass?, 102
    compatibleSuperclasses?, 102
    computeAllocator, 108
    computeAndEnsureFieldAccessors, 101, 104
    computeConstructor, 108
    computeCPL, 103
    computeDiscriminatingFunction, 111
    computeFieldReader, 105
    computeFieldWriter, 105
    computeInheritedFields, 103
    computeInheritedInitializer, 108
    computeInheritedKeywords, 103

computeInitializer, 108
computeInstanceSize, 109
computeKeywords, 103
computeMethodLookupFunction, 111
computeNewField, 103
computeNewFieldClass, 103
computePrimitiveReaderUsingClass, 106
computePrimitiveReaderUsingField, 106
computePrimitiveWriterUsingClass, 106
computePrimitiveWriterUsingField, 106
computeSpecializedField, 103
computeSpecializedFieldClass, 103
domain:, 110
ensureFieldReader, 105
ensureFieldWriter, 105
initFunction:, 100
initialize, 107
initKeyword:, 100
initValue:, 100
methodClass:, 110
methods:, 110
name:, 100
primitiveAllocate, 109
reader:, 100
requiredInitKeyword:, 100
setPrimitiveClassOf, 109
specifiedFields:, 100
specifiedKeywords:, 100
specifiedSuperclasses:, 100
type:, 100
writer:, 100
Multimethoden, 80

Nachrichtenaustausch, 35
Nachrichtenversenden, 35, 156
Netze
semantische, 26

OakLisp, 49
Oberon, 38
Objekt, 31, 75, 289
-Allokation, 83
-Erzeugung, 274
-Initialisierung, 83, 159, 274
-Protokoll, 80
-Struktur, 75, 76
-Verhalten, 75, 80
Allokation, 235
Eigenschaft, 75
erster Ordnung, 129, 289
Erzeugung, 60
funktionales, 71, 112
Initialisierung, 60, 288
initialisierung, 235
Objektebene, 63, 155, 197, 224, 226, 273
Objekteigenschaften, 75
instanzspezifisch, 75
klassenspezifisch, 75
Objekterzeugung, 159
Objekthierarchie, 290
objektorientierte
Konzepte, 114
Sprachaspekte, 114
Objektsystem
einstufig, 31, 34
zweistufig, 31
OBJTALK, 51
ObjVLisp, 39, 51, 151
Operation, 80
Funktion, 80
Lese-, 59, 77, 134
Prozedur, 80
Schreib-, 59, 77, 134
Optimierungen, 176
dynamische, 125
Last-Call-, 123
Restrekursionsauflösung, 123
Speicherplatz-, 123, 143
statische, 124
Typinferenz, 123
zur Ladezeit, 123
zur Laufzeit, 123
Overriding, 34, 274

Package, 74
parent, 34
Performanz, 176, 187, 211
TELOS, 246
Persistenz, 73

PMFS, 49
Polymorphie, 36
Primärklassen, 94
Primärmethode, 38
Programmiersprachen
    Assembler, 26
    constraint-orientierte, 27
    deklarative, 27
    echtzeit-fähige, 27
    funktionale, 27
    höhere, 26
    imperative, 27
    maschinennahe, 26
    objektorientierte, 27
    parallele, 27
    prozedurale, 27
    regel-orientierte, 27
Programmierstil, 27, 28
    objektorientierter, 29
Protokoll, 80
    Allokations-, 107, 243
    Feldzugriffs-, 104, 240
    generischer Dispatch, 245
    Initialisierungs-, 65, 107, 243
    Introspektions-, 65, 98, 238
    Methodenauswahl, 245
    Slotzugriffs-, 66
    Spracherweiterungs-, 97
    Vererbungs-, 65, 102

Quellcode
    -Interpretation, 145

rapid prototyping, 15
Rechnerarchitektur
    von Neumann-, 27
Redefinieren, 61
    von Klassen, 61, 135, 262
Reflektion, 19, 97
Reflektion in Programmiersprachen, 38
reflektive Systeme, 38
Reklassifizieren, 62
    von Instanzen, 62
    von Objekten, 264
Restparameter, 144
Robustheit, 21

Scheme, 48
Schichtenmodell, 119
Self, 31
semantische Netze, 26, 34
Sendeereignis, 35
Senden von Nachrichten, 35
Shadowing, 274
shared slots, 34, 79
Sichtbarkeit, 73
Signatur, 290
    kongruente, 81
Simula, 26, 33
Singleton-Methoden, 76, 82
Slot, 76, 290
slot-value-using-class, 107
Slotzugriff, 159
Slotzugriffsprotokoll, 66
Smalltalk, 26, 33, 39, 40, 68
Software-Engineering, 8
Softwarelebenszyklus, 13
Softwaresystem
    Design von, 14
Spezialisieren, 34
    von Feldannotationen, 274
    von Objektklassen, 85
    von Struktur, 87, 91
    von Verhalten, 88, 92
Spezifizität
    von Methoden, 82
Spiralenmodell, 13
Spracheinbettung, 8
Spracherweiterungsprotokoll, 97
Sprachkonzepte, 69, 279
statische Struktur, 75
statisches Verhalten, 75
Struktur
    dynamisch, 75
    instanzspezifisch, 75
    klassenspezifisch, 75
    statisch, 75
    von Objekten, 76
subclass, 34
Subclassing, 34
Subklasse, 290
    direkte, 287
    indirekte, 288
Subsumtion, 290

super(), 38
superclass, 34
Superklasse, 290
    direkte, 86, 287
    indirekte, 288
System
    Entwicklungs-, 123
    Laufzeit-, 123
Systemarchitektur, 131

TELOS, 219, 221, 222, 271, 277
TINA, 12, 222, 271

Verarbeitungsmodell
    formales, 27
    kognitives, 27
Verbmobil, 9
Vererbung, 33, 55, 76, 240, 273, 290
    einfache, 35, 85, 86
    Mixin-, 35, 93, 203
    multiple, 35, 85, 90
Vererbungsgraph, 86
Vererbungsprotokoll, 65, 102
Verhalten
    dynamisch, 75
    instanzspezifisch, 75
    klassenspezifisch, 75
    statisch, 75
    von Objekten, 80
Verteilte Systeme, 72
Verursacherprinzip, 23
virtual function, 90
virtuelle Maschine, 27

Wartbarkeit, 31
Wasserfallmodell, 13
week pointer, 139
WFS, 151
Wiederverwendbarkeit, 31

Zustand
    instanzspezifisch, 76
    klassenspezifisch, 76