Seminar "Objektorientiertes Programmieren"
Theorie, Methodik und Anwendung

Editor: N. Schmeißer[1]
Forschungszentrum Rossendorf e.V.,
Abt. Kommunikation und Datenverarbeitung

21. September 1998

[1] Telefon: (+49 351) 260/2207, E-Mail: nils@schmeisser.com,
URL: http://www.fz-rossendorf.de/FVTK/MITARB/schmei

Inhaltsverzeichnis

1 Einführung **4**
- 1.1 Zielstellung . . . 4
- 1.2 Einführung . . . 5

2 Theoretische Aspekte **8**
- 2.1 Begriffe und Prinzipien . . . 8
  - 2.1.1 Objekte, Klassen und Vererbung . . . 8
  - 2.1.2 Abstrakter Datentyp . . . 9
  - 2.1.3 Implizite Typkonversion und Polymorphismus . . . 11
- 2.2 Das Klassenkonzept . . . 13
  - 2.2.1 Informale Beschreibung . . . 13
  - 2.2.2 Theorie der Objektorientierten Programmierung . . . 22

3 OO-Sprachen **24**
- 3.1 Turbo-Pascal . . . 24
- 3.2 Oberon . . . 28
  - 3.2.1 Überblick . . . 28
  - 3.2.2 Die Sprache Oberon-2 . . . 28
  - 3.2.3 Das Oberon - System . . . 36
  - 3.2.4 Hinweise zum Umgang mit Oberon . . . 36
  - 3.2.5 Erfolg/Ausblick . . . 37
- 3.3 Smalltalk . . . 39
  - 3.3.1 Die Geschichte von Smalltalk und Smalltalk's Big Ideas . 39
  - 3.3.2 Objekte und Nachrichten . . . 41
  - 3.3.3 Klassen, Instanzen und Methoden . . . 41
  - 3.3.4 Zur Syntax von Objekten und Nachrichten . . . 42
  - 3.3.5 Klassen- und Methodenimplementation . . . 45
  - 3.3.6 Vererbung . . . 46
  - 3.3.7 Eine Binärrelation als Beispiel . . . 47
  - 3.3.8 Internas und Implementation . . . 48
- 3.4 C++ . . . 51
  - 3.4.1 Allgemeine Einführung . . . 51

3.4.2 Paradigmen des Programmierens und ihre Realisierung in C++ . . . 52
3.4.3 Unterstützung für Datenabstraktion . . . 55
3.4.4 Unterstützung des objektorientierten Programmierens . . 58
3.4.5 Praktische Hinweise . . . 60
3.5 Java . . . 61
3.5.1 Ursprung und Eigenschaften . . . 61
3.5.2 Einfache Datentypen und Konstrukte . . . 62
3.5.3 Objektorientierte Programmierung mit Java . . . 65
3.5.4 Eigene Erfahrungen . . . 69
3.6 Ada 95 . . . 70
3.7 Eiffel . . . 74
3.8 FORTRAN 95/FORTRAN 2000 . . . 76
3.8.1 Introduction . . . 76
3.8.2 Object Oriented Programming in Fortran 2000 . . . 79
3.8.3 Procedure Pointer . . . 84

4 OO-Systeme 87
4.1 Graphische Nutzeroberflächen (GUI) . . . 87
4.2 Verteiltes Objektorientiertes Programmieren . . . 90
4.2.1 Common Object Request Broker Architecture . . . 90
4.2.2 DCOM . . . 91
4.2.3 Java Beans/RMI . . . 91
4.2.4 OOMPI . . . 92
4.3 Datenbanken . . . 93
4.3.1 Relationale Datenbanken . . . 93
4.3.2 Semantisches Datenbankmodell . . . 94
4.3.3 Objektorientierte Datenbanken . . . 94

A Nutzung der Systeme 100
A.1 Turbo-Pascal . . . 100
A.2 Oberon . . . 101
A.3 Smalltalk-80 . . . 103
A.4 C++ . . . 105
A.5 Java . . . 106
A.6 Ada 95 . . . 107
A.7 Eiffel . . . 107

B Übungen 109
B.1 Euklidische Geometrie . . . 109
B.2 Verfahren und Methoden der Algebra . . . 110
B.2.1 Algebraische Strukturen (Mengen, Gruppen, Ringe, Körper) 110
B.2.2 Vektoren und Matrizen . . . 110
B.2.3 Taylorarithmetik . . . 110

B.3 Formelmanipulation 111
B.4 attributierte Graphen 111
B.5 Grafisches Basissystem 113
B.6 minimales Datenbanksystem 113
B.7 Beispiele 114

C Beispiele 115

C.1 Turbo-Pascal 115
C.2 Oberon 120
C.3 Smalltalk-80 128
C.4 C++ 133
C.5 Java 139
C.6 Ada 95 143
C.7 Eiffel 148

D Autoren 154

E Bild- und Literaturnachweis 155

F Bild- und Literaturnachweis 162

Kapitel 1

Einführung

Inhalt

Aufbauend auf dem vorliegenden Script wurde im Sommersemester 1997 am Institut für wissenschaftliches Rechnen der TU Dresden in Zusammenarbeit mit dem Rechenzentrum des Forschungszentrums Rossendorf ein Seminar mit dem Titel "Objektorientiertes Programmieren" gehalten. Dieses Script stellt weiterhin die Vereinigung aller dort gehaltenen Vorträge dar.
Wir erläutern die Zielstellung des Seminars und werden im zweiten Abschnitten einen kurzen Überblick über die Entwicklung von Programmiersprachen geben.

1.1 Zielstellung

Das Ziel des Seminares besteht darin, Sie mit den Grundlagen des objektorientierten Programmierens vertraut zu machen und Sie in die Lage zu versetzen das Dargebotene an praktischen Problemen anwenden zu können.
Die am Schluß angegebenen Aufgaben überdecken einen Großteil praxisrelevanter Probleme. Es wird empfohlen B.1 und B.2 vollständig abzuarbeiten, B.3 bis B.6 stellen größere Projekte dar, die aber im Prinzip alle nach dem selben Schema lösbar sind (Objekt-/ Klassenhierarchie definieren, dann Erzeugung von Instanzen und deren Verwaltung). Von diesen sollten Sie mindestens eines vollständig bearbeiten.
In den nachfolgenden Abschnitten werden wir uns zunächst mit einigen grundlegenden Aspekten des objektorientierten Programmierens beschäftigen. Danach werden wir uns verschiedenen OO-Programmiersprachen zuwenden. Wir werden keine vollständige Beschreibung der Sprache geben, sondern uns auf die wesentlichen, zum objektorientierten Programmieren gehörenden Dinge beschränken.

Den Schluß bilden Ausführungen zu Software-Systemen, die Prinzipien des OOP übernehmen bzw. in einer OO-Sprache geschrieben wurden.

1.2 Einführung

An aller erster Stelle soll hier natürlich die Frage stehen: Warum betreibt man überhaupt OOP.
Die Entwicklung von Programmiersprachen vollzog sich folgerichtig von der primitiven Aneinanderreihung von Binär-Codes (Assembler) hin zu Hochsprachen, die in ihrem ersten Stadium keinerlei Struktur kannten. Diese Sprachen waren zwar relativ einfach zu erlernen (BASIC = Beginners All Purpose Symbolic Instruction Code), aber im Nachhinein begann man zu erkennen, daß die Implementation noch nicht einmal besonders raffinierter Algorithmen sehr schwierig, wenn nicht sogar unmöglich war. So lassen BASIC und FORTRAN 77 beispielsweise keine Rekursion zu. Die, in diesen alten Sprachen geschriebenen Programmquelltexte waren darüber hinaus in den meisten Fällen sehr unleserlich und es gab eine Menge Möglichkeiten Fehler zu machen. Ein erster Schritt in die richtige Richtung wurde mit der Einführung des prozeduralen Programmierens getan. Im Gegensatz zum vorangegangenen, rein iterativen Stil hat man hier schon die Möglichkeit Subalgorithmen als eigenständige Einheiten zu formulieren und sogar einzeln zu testen. Insbesondere von Interesse ist hier die Verwendung **lokaler Variablen**, die in ihrem Zustand nur innerhalb der jeweiligen Struktureinheit geändert werden können (→ *Block-Konzept*). Mit der Einführung *modularer Sprachen* konnten, mit der Verwendung globaler Variablen verbundene Probleme etwas besser kontrolliert werden[1] (C, Modula). Parallel dazu setzte sich immer mehr die Verwendung eines strengen Typ-Konzeptes durch (Pascal, Modula). Allen diesen Sprachen war bis jetzt gemein, das die Funktionalität im Vordergrund stand (imperativer Programmierstil). So ist es auch nicht weiter verwunderlich, das Myriaden von Funktionen und Prozeduren geschrieben wurden, um alle eventuell auftretenden Daten behandeln zu können. Die Wiederverwendbarkeit von Code ist zwar möglich (CERN, NAG Bibliotheken) jedoch nur solange sich die zu bearbeitenden Daten innerhalb eines gewissen Bereiches bewegen.
Eine völlig andere Herangehensweise wurde mit dem objektorientierten (OO) Programmierstil geschaffen. Hier stehen die Daten als Objekte der Begierde im Mittelpunkt. Über diesen (man beachte die schon fast mathematische Terminologie) werden dann erst die Operationen, Prozeduren und Funktionen erklärt. Die Wiederverwendbarkeit ist natürlich auch hier gegeben, nämlich genau über der selben *Klasse* von Daten. Damit ist es nicht mehr möglich, eine *Methode* auf eine völlig falsche Menge von Eingabedaten anzuwenden, was leider bei nicht

[1] Eigentlich gibt es keinen vernünftigen Grund, der die Verwendung einer globalen Variablen erzwingt. In einigen wenigen Ausnahmefällen kann die Verwendung bezüglich eines Moduls globaler Variablen sinnvoll sein (Zähler).

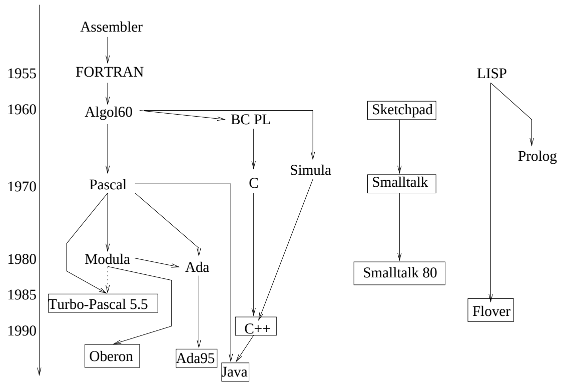


Abbildung 1.1: Stammbaum von Programmiersprachen

objektorientierten-Sprachen noch allzu häufig der Fall ist. OO Sprachen vereinen in sich die Möglichkeit der Datenkapselung (dies dient dem Schutz des Programmierers vor sich selbst und vor anderen). Ein wesentlicher Bestandteil einer jeden OO-Sprache ist der Vererbungsmechanismus, mittels dessen es ohne weiteres möglich ist, die Funktionalität einer gegebenen Implementation an eine Unterklasse weiterzureichen und dabei zu verfeinern oder zu ersetzen (→ *Spezialisierung*). Zusätzlich bieten OO-Sprachen die Möglichkeit, über, voneinander verschiedenen Klassen, Methoden mit dem selben Namen zu erklären (z.B. eine Methode "Dimension" für Vektoren, Punkte, Matrizen). Man nennt dieses Konzept *Polymorphie*.
Betrachtet man den Stammbaum der Programmiersprachen, so fällt die Entwicklungslinie von *Smalltalk 80* über *Smalltalk* und *Sketchpad* und deren zeitiger Ursprung natürlich auf. Smallktalk 80 stellt die konsequente Weiterenwicklung einer OO-Sprache gepaart mit dem Versuch einer nutzerfreundlichen Oberfläche (→ GUI) dar. Weiterhin fällt die Ballung der OO Erweiterungen der Pascal/C Sprachfamilie ins Auge. Zwar gibt es mit FORTRAN 90 den Versuch auch dieser recht alten Sprache neues Leben einzuhauchen, jedoch dürfte in nächster Zeit nicht viel davon zu hören sein. Jüngste Entwicklung unter den OO-Sprachen ist nach Oberon (diese von eher akademischen Interesse) Java. Sowohl Java als auch Smalltalk übersetzen den Hochsprachencode zunächst mittels eines Compilers in Byte-Code, der dann interpretativ von einer virtuellen Maschine ausgeführt wird[2]. Im Gegensatz dazu erzeugen C++, Ada und die Borland-Pascal Familie direkt ausführbare Maschinenprogramme und haben damit natürlich einen Geschwindigkeitsvorteil.

[2] Das ist kein Manko der Sprache, das ist einfach ihr Konzept (siehe auch [Sun95] und [Hof87]). Sun hat bereits Prozessoren angekündigt, die Java-Bytecode direkt ausführen können.

Einher mit der Entwicklung objektorientierter Sprachen ging die Entwicklung immer komplexerer und leistungsfähigerer Software.
Wir wollen hier zunächst auf grafische Oberflächen verweisen (X-Windows (M.I.T.), NeWS (Sun), Microsoft Windows, OS/2 Presentation Manager (IBM), die Macintosh Oberfläche und die grafische Oberfläche des Comodore Amiga).
Von den aufgeführten Systemen verwenden allerdings nur MS-Windows ab Version 3.1, die Macintosh Oberfläche, der Presentation Manager und die Amiga Oberfläche objektorientierte Mechanismen.
Neben diesen Anwendungen konnten große Datenbanksysteme erst durch die Verwendung objektorientierter Sprachen bzw. der Verwendung beim OO Programmieren gültiger Prinzipien entstehen.
In letzter Zeit tauchen immer häufiger Überlegungen zum verteilten objektorientierten Programmieren auf (CORBA, DCOM). Diese Gedanken entspringen zum Großteil dem Bedarf, verteilte Systeme zu verwenden (Datenbanken, Ressourcenteilung, Prozeßsimulation) und ein Maximum an Rechenleistung zu erzielen ("Grand Challenge" Probleme).

Kapitel 2

Theoretische Aspekte

2.1 Begriffe und Prinzipien

2.1.1 Objekte, Klassen und Vererbung

Formal spricht man von einem *Objekt* als

- eingenständigem Individuum,
- mit charakteristischen Eigenschaften,
- inneren Zuständen,
- Attributen und
- Methoden.

Diese Merkmale eines Objektes können am Beispiel des Kreises veranschaulicht werden.
Jeder Kreis ist offensichtlich ein eigenständiges Individuum. Er kann geometrischen Transformationen unterworfen und kann visualisiert werden. Typischerweise wird ein Kreis als die Menge aller der Punke charakterisiert, die zu einem vorgegebenen Punkt (Mittelpunkt) den gleichen Abstand (Radius) haben. Ein Kreis befindet sich immer an einer bestimmten Position (innerer Zustand). Diese in den Attributen abgelegte Informationen über den Zustand kann durch eine Transformation (Methode) geändert werden.

Beispiel: `k.draw` um einen Kreis zu zeichnen
`k.moveto(x,y)` um einen Kreis zu verschieben

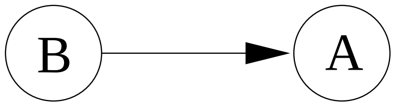


Abbildung 2.1: Ableitungsschema nach [Mey89]

Der Aufruf einer Methode wird auch als Versenden einer Nachricht an das Objekt gedeutet [Hof87]. Die Methode[1] interpretiert die übergebenen Parameter und überführt den ursprünglichen Zustand des Objektes in einen neuen.
Die Abstraktion vom konkreten Objekt führt zum Begriff der *Klasse* als Vereinigung aller Objekte mit gleichen Eigenschaften. Ein aus einer Klasse generiertes Objekt heißt *Objekt der Klasse* oder *Instanz der Klasse* [AC96]. Eine Klasse besitzt immer eine Methode, um eine Instanz zu erzeugen (*Konstruktor*).
Durch Verändern (*Overriding*) oder Hinzufügen von Attributen oder Methoden zu einer Klasse wird aus dieser eine neue Klasse erzeugt. Diese neue Klasse heißt *abgeleitete Klasse*. Die Vererbung auf neue Klassen definiert eine transitive *Ableitungsrelation*. [Mey89] beschreibt Ableitungsrelationen zwischen Klassen durch einen Pfeil. Abbildung 2.1 stellt B als von A abgeleitete Klasse dar. A heißt auch *Superklasse* von B.
Weitere gängige Notationen sind "**sublcass** B **of** A" [AC96] oder einfach in der C++-Notation "A::B" [Str93].
Es existieren verschiedene Formalisierungen der Begriffe Objekt, Klasse und Vererbung, die in den folgenden Abschnitten erläutert werden sollen. Das Basiskonzept der erläuterten Kalküle bildet der *abstrakte Datentyp* [GM94].

2.1.2 Abstrakter Datentyp

[GM94] führt zur Beschreibung abstrakter Datentypen zunächst ein Schema $i : \tau$, bestehend aus einem Identifikator i und einem Typbezeichner τ ein.

Definition 2.1 :
Seien $i_1, ..., i_n$ Bezeichner mit zugehörigen Typen $\tau_1, ..., \tau_n$, dann wird durch die Komposition

$$\omega = i_1 : \tau_1 + ... + i_n : \tau_n \tag{2.1}$$

der Typausdruck ω erklärt. ω wird auch *Sorte* genannt.

$n = 1$ korrespondiert genau mit dem unstrukturierten Datentyp, $\tau_i = \tau \forall i$ liefert das Feld. Alle restlichen Fälle ergeben den *strukturierten Datentyp*, der auch einfach *Struktur* oder *Record* genannt wird [2].

[1] Eine Methode wird in diesem Sinne als Teil der Nachricht verstanden. Sie hat eine vorbestimmte Semantik.

[2] Ein Feld kann als strukturierter Datentyp mit den Komponenten $x_1 : \tau + ... + x_n : \tau$ aufgefaßt werden kann.

Beispiel: Es seien die Typen $REAL$ (reelle Zahlen) und $CHAR$ (Zeichen) gegeben.
$c : CHAR$ bezeichnet eine Variable vom Typ $CHAR$.
Das Feld $s : (c_0 : CHAR + ... + c_{255} : CHAR)$ kann auch als eine Instanz der Sorte $STRING = c_0 : CHAR + ... + c_{255} : CHAR$ aufgefaßt werden ($s : STRING$). Die Komponente $c_k : CHAR$ korrespondiert hierbei zum Element $s[k]$ des Feldes s.
Auf die gleiche Art und Weise wird die Struktur

```
Kreis is
    x:REAL
    y:REAL
    name:STRING
end Kreis
```

durch die Sorte
$KREIS = x : REAL + y : REAL + name : STRING$

beschrieben.

Zwei Sorten ω_1 und ω_2 sollen gleich sein ($\omega_1 \cong \omega_2$), falls

$$\exists \pi \in \Pi | \omega_1 = \pi(\omega_2) \tag{2.2}$$

gilt. $\pi(\omega)$ permutiert die Komponenten von ω.
Über einer Menge $\Omega = \{\omega_i\}$ von Sorten kann jetzt eine Menge von $n-$stelligen Operatoren $\delta : \Omega^n \to \Omega \in \Delta_n$ erklärt werden. Jedem Operator wird eine *Spezifikation* $\Gamma_\delta = ((\omega_{i_1}, ..., \omega_{i_n}), \omega) \in \Omega^n \times \Omega$ zugeordnet.
Sei nun B eine homomorphe Abbildung $B : \Omega \to S$, die jeder Sorte ihre Semantik S zuordnet, so entsteht unter Einbeziehung einer *Interpretation* $\gamma_\delta \in B(\omega_{i_1}) \times ... \times B(\omega_{i_n}) \to B(\omega)$, $\forall \delta \in \Delta_n$ eine $\Omega\Delta\Gamma$-*Algebra* $\mathcal{A}$. B als Ω-indizierte Familie von Mengen $(B(\omega))$ heißt Träger der Algebra [GM94][3].

[3]Sei x eine Variable vom Typ $\omega = i_1 : \tau_1 + ... + i_n : \tau_n$ (also $x \in (\tau_1 \times ... \times \tau_n)$), dann liefert $B(x)$ den Wert von x; $B(x) = B(x.i_1) \times ... \times B(x.i_n)$

Definition 2.2 :

Gegeben sei eine Menge Ω von Sorten, eine Menge Δ von Operatoren über Ω und deren Spezifikation Γ zusammen mit einer homomorphen Abbildung $B : \Omega \to S$ in eine Menge semantischer Ausdrücke, dann heißt die mehrsortige $\Omega\Delta\Gamma$-Algebra $\mathcal{A}$ *abstrakter Datentyp*.

Beispiel: Der abstrakte Datentyp

- $\Omega = \{IMAG, BOOL\}$; $IMAG = a : REAL + b : REAL$, $BOOL = w : BOOLEAN$
- $B(IMAG) = a + \hat{i}\ b$, $B(BOOL) = w$
- $==: (IMAG, IMAG) \mapsto BOOL$
  $\Gamma_{==} = ((IMAG, IMAG), BOOL)$
  $\gamma_{==} : (a_1 + \hat{i}\ b_1, a_2 + \hat{i}\ b_2) \mapsto (a_1 == a_2) \wedge (b_1 == b_2)$
  $+ : (IMAG, IMAG) \mapsto IMAG$
  $\Gamma_{+} = ((IMAG, IMAG), IMAG)$
  $\gamma_{+} : (a_1 + \hat{i}\ b_1, a_2 + \hat{i}\ b_2) \mapsto (a_1 + a_2) + \hat{i}\ (b_1 + b_2)$
  $- : (IMAG, IMAG) \mapsto IMAG$
  $\Gamma_{-} = ((IMAG, IMAG), IMAG)$
  $\gamma_{-} : (a_1 + \hat{i}\ b_1, a_2 + \hat{i}\ b_2) \mapsto (a_1 - a_2) + \hat{i}\ (b_1 - b_2)$

beschreibt die additive Untergruppe der komplexen Zahlen.

$(\{(\omega_{i_1}, ..., \omega_{i_k}) = \omega_{i_1} + ... + \omega_{i_k} : \omega_i \in \Omega, k \in \mathbf{N}\}, +)$ ist isomorph zur Permutationsgruppe über Ω. (das ist gleich dem Träger von Δ). Für alle Elemente $\omega = \omega_{i_1} + ... + \omega_{i_k}$ von Ω ist dann $B(\omega) = \times_{i=1}^{l(\omega)} B(\omega_i)$; $l(\omega) = k$. Im Folgenden soll Ω diese Gruppe bezeichnen.

2.1.3 Implizite Typkonversion und Polymorphismus

Im vorhergehenden Abschnitt wurde die Konversionen zwischen den generischen Komponententypen τ außer Acht gelassen. [GM94] führt hierfür Algebren mit geordneten Sorten ein.

Definition 2.3 :

Existiert für zwei Sorten ω und ω' einer $\Omega\Delta\Gamma$-Algebra ein Homorphismus $\omega <: \omega'$ so, daß

$$\begin{aligned} &B(\omega <: \omega') \in B(\omega) \to B(\omega'); \\ &B(\omega <: \omega) = id; \\ &\omega <: \omega' \wedge \omega' <: \omega'' \Rightarrow B(\omega <: \omega'') = B(B(\omega <: \omega') <: \omega'') \end{aligned} \tag{2.3}$$

so heißt ω ein *Subtyp* von ω'.

$<:$ heißt *implizite Typkonversion*. Damit wird jede Abbildung, die über ω' arbeitet, zu einer polymorphen Abbildung über ω. Aufgrund der Definition folgt sofort

$$B(\Gamma_\delta(..., \omega, ...)) \to B(\Gamma_\delta(..., \omega', ...)) \tag{2.4}$$

Beispiel: Es ist $v : INTEGER <: r : RATIONAL$ mit
$r : RATIONAL = z : INTEGER + n : INTEGER_{\setminus\{0\}}$

$B(\omega <: \omega')$:
$B(v : INTEGER) \rightarrow$
$B(v : INTEGER + 1 : INTEGER) \forall v$

$B(\omega <: \omega)$:
$B(v : INTEGER + 1 : INTEGER) = \frac{v}{1} = v$
$= B(v : INTEGER) \Rightarrow$
$B(v : INTEGER <: v : INTEGER) = id$

Diese Ausprägung des Polymorphismus wird *Ad-hoc Polymorphismus* oder *Subtyppolymorphismus* genannt [GM94].
Die Annahme

$$a : A \wedge A <: B \Rightarrow a : B \tag{2.5}$$

charakterisiert eine spezielle, "subsumption" genannte Eigenschaft der Subtyprelation. 2.5 sagt, daß ein Wert vom Typ A auch als ein Wert vom Typ B betrachtet werden kann [AC96].
Sei der Homomorphismus κ der transitive Abschluß von $<:$ über $\Omega \times \Omega'$, so folgt nach dem Homomorphiesatz für Gruppen

$$\Omega' | kern(\kappa) \cong \Omega \tag{2.6}$$

und damit

$$\delta : \omega \rightarrow \Omega = \Omega' | kern(\kappa) \rightarrow \Omega' | kern(\kappa) \tag{2.7}$$

Ein Operator δ kann deshalb auch durch einen Operator über Ω' gemäß
$\delta = \kappa \circ \delta'|_{\Omega' | kern(\kappa)}$ ausgedrückt werden. Das bedeutet natürlich

$$B((\kappa \circ \delta')(\omega)) = B(\delta(\omega)) \tag{2.8}$$

Definition 2.4 :
Seien $\mathcal{A}$ und $\mathcal{A}'$ $\Omega\Delta\Gamma - Algebren$ mit $\Omega \subseteq \Omega'$ und existiere eine Sortenkonvertierung $\kappa : \Omega' \rightarrow \Omega$, dann heißen zwei Operatoren $\delta \in \Delta$ und $\delta' \in \Delta'$ *polymorphe Operatoren über* Ω *und* Ω', wenn gilt

$$\kappa \circ \delta'|_{\Omega' | kern(\kappa)} \equiv \delta \tag{2.9}$$

$\mathcal{A}$ heißt *von* $\mathcal{A}'$ *abgeleiteter abstrakter Datentyp* ($\mathcal{A} <: \mathcal{A}'$), wenn außerdem gilt

$$\forall \delta \in \Delta \Rightarrow \exists \delta' \in \Delta' : \kappa \circ \delta'|_{\Omega' | kern(\kappa)} \equiv \delta \tag{2.10}$$

Wegen (2.8) folgt für polymorphe Operatoren $B(\delta'(\Omega)) = B(\delta(\Omega))$. Diese zweite Form der Polymorphie heißt *universeller Polymorphismus*.

2.2 Das Klassenkonzept

Objekte und Klassen sind wesentlich komplexere Strukturen als abstrakte Datentypen. Um die durch das Klassenkonzept gebenen Möglichkeiten voll erfassen und nutzen zu können, ist sowohl eine genaue Beschreibung der Semantik von Objekten, als auch die Festlegung der zu ihrer Beschreibung nötigen syntaktischen Strukturen nötig. Die Klassen oder Objekte beschreibenden Konstrukte sind in der Regel selbsterklärend und können deshalb zunächst weniger formal beschrieben werden.
Um die Bergiffe und Prinzipien des objektorientierten Programmierens exakt formulieren zu können ist die Angabe eines geeigneten Kalküls sinnvoll und nützlich [AC96]. Aus den in den folgenden Abschnitten beschriebenen Kalkülen können im Weiteren Anforderungen an die Syntax einer Beschreibungssprache abgeleitet werden, die geeignet ist, ein Laufzeittypinformationssystem zu unterstützten.

2.2.1 Informale Beschreibung

Klassenbasierte Beschreibung

Zur Definition von Klassen wird in einer Reihe objektorientierte klassenbasierter Sprachen das folgende Schema verwendet [AC96], [Str93], [AG96] und [Mey92]

```
class classname is
   var attribute : typeidentifier := initialvalue ;
   method methodname ( argumentlist ) : returntype is
      methodbody
   end ;
end ;
```

Mittels dieser Notation ist es möglich, sämtliche Attribute und Methoden einer Klasse zu erklären. [AC96] bezeichnet zur Vereinfachung auch Methoden als Attribute.
class impliziert die Existenz eines formalen Attributes **self** vom Typ *InstanceTypeOf(classname)*, welches auf die "eigene" Realisierung verweist. Eine Instanz einer Klasse wird durch das **new** -Konstrukt erzeugt.

```
var instance : InstanceTypeOf (classname) := new classname ;
```

Für eine Programmvariable `instance` werden zwei unterschiedliche Interpretationen verwendet. Zum einen kann eine Variable ein Objekt an sich bezeichnen, im anderen Fall repräsentiert sie eine Referenz auf ein Objekt. C++ und Oberon unterscheiden diese beiden Arten der Interpretation, während Simula, Smalltalk und Modula-3 nur die Interpretation als Referenz verwenden. [AC96] veranschaulicht diese Paradigmen in einem "naiven Speichermodell" (Abbildung 2.2).

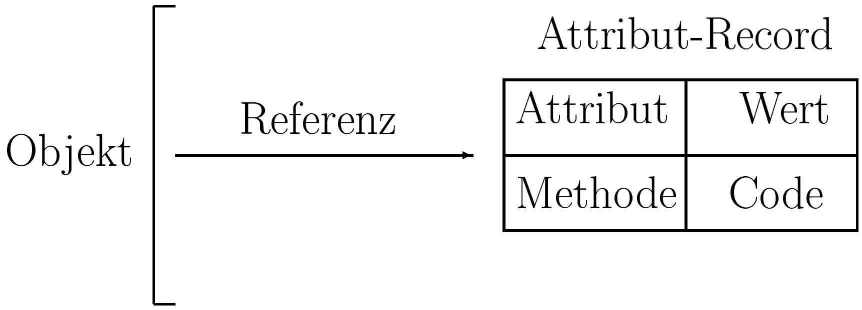


Abbildung 2.2: "Naives Speichermodell" nach [AC96]

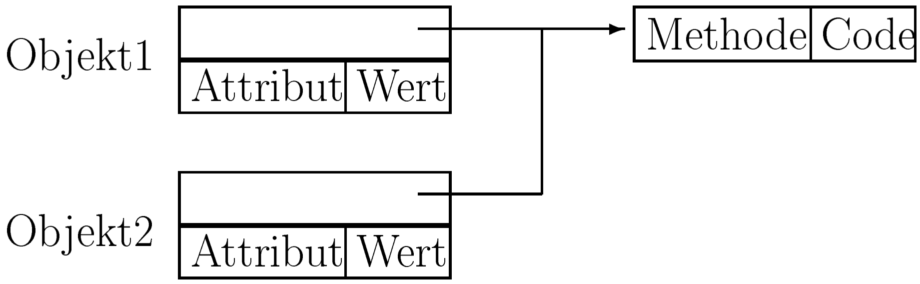


Abbildung 2.3: Methodenaufruf und Methodensammlung

Ebenso wie der Zugriff auf ein Attribut erfolgt auch der Aufruf einer Methode über eine Qualifizierung in der Form

```
object . methodname ( parameterlist )
```

In klassenbasierten Sprachen sind Methoden nicht direkt in ein Objekt eingebunden [4]. Vielmehr delegiert ein Objekt den Aufruf an eine, ihrer Klasse zugeordnete Methoden-Suite. Normalerweise wird hierbei der Wert von **self** implizit an die gerufene Methode weitergeleitet. In Oberon muß **self** explizit als Argument vom Typ $InstanceTypeOf(classname)$ an die Methode weitergereicht werden [RW92].
Abbildung 2.3 stellt den Vorgang des Methodenaufrufes über eine Methodensammlung dar.
Wie schon im Abschnitt 2.1.1 erläutert, entsteht durch Hinzufügen oder Überschreiben von Attributen einer Klasse eine abgeleitete Klasse. Das hier erläuterte Schema gestattet auch die Formulierung des Vererbungsvorganges.

[4] C++ bietet ein spezielles Konstrukt an, das eine Einbindung erzwingt.

```
subclass subclassname of classname is
    var  newattribute : typeidentifier := initialvalue ;
    override methodname ( argumentlist ) := returntype is
        methodbody
        self .newattribute := self .attribute;
        super .methodname ( ... );
    end ;
end ;
```

super kennzeichnet einen Schlüssel, der aus der neuen Methode heraus den Zugriff auf die überschriebene Methode gestattet. Damit können hierarchische Methodensammlungen implementiert werden, die **super** als Verweis auf ihren direkten Vorgänger enthalten. Im Fall der mehrfachen Vererbung, der durch

subclass `subclassname` **of** $\texttt{class}_1\texttt{,...,class}_n$ **is** ...

ausgedrückt wird, kann die Interpretation von **super** problematisch werden.

$$InstanceTypeOf(c') <: InstanceTypeOf(c) \Leftrightarrow c' :: c \tag{2.11}$$

induziert zusammen mit dem Subtypbegriff aus 2.1.3 zwei Schemata zur Interpretation des Aufrufes von `x.mname(...)` im nachfolgenden Programmtext. Sei $sc :: c$ und

```
var  i : InstanceTypeOf(c) := new c;
var  is: InstanceTypeOf(sc) := new sc;
procedure f(x:InstanceTypeOf(c)) is
    x.mname(...);
end ;
...
f(is);
```

dann ruft `x.mname` bei

statischer Bindung : die Methode `mname` der Klasse `c`

dynamischer Bindung : die Methode `mname` der Klasse `c'`

Bei *dynamischer Bindung* wird also Code in Abhängigkeit von $TypeOf(x)$ ausgeführt. Statische Bindung wird durch die *Typreduktion* 2.12 (*Subsumption*) motiviert [AC96].

$$a : A \wedge A <: B \Rightarrow a : B \tag{2.12}$$

Mitunter verfügen OO-Sprachen über ein Typinformationssystem (z.B. in Modula-3, Simula und Oberon), mit dessen Hilfe es möglich ist, eine pseudo-dynamische Bindung zu generieren. Der folgende Code demonstriert die Verwendungen von Typinformationen:

```
typecase x
    when i:InstanceTypeOf(c) do i.mname(...);
    when is:InstanceTypeOf(cs) do is.mname(...);
end ;
```

Obwohl dieser Ansatz das Prinzip der Datenabstraktion im Sinne der objektorientierten Programmierung unterläuft ist die Existenz eines Typinformationssystems von großem Nutzen für die Analyse aller Instanzen und ihrer Zusammenhänge.
Während konservative Ansätze das Überschreiben von Methoden nur durch Methoden gleichen Typs zulassen, ist unter Einhaltung der nachfolgenden Regeln eine *Spezialisierung* während des Überschreibens möglich.

```
class c is
    method m(x:A):B is ... end ;
    method m1(x:A1):B1 is ... end ;
end ;
subclass c' of c is
    method m(x:A'):B' is ... end ;
end ;
```

Wegen $c' :: c$ muß natürlich auch $InstanceTypeOf(c') <: InstanceTypeOf(c)$ gelten. Nach der Reduktion eines Objektes o' vom Typ $InstanceTypeOf(c')$ zu $InstanceTypeOf(c)$ muß beim Aufruf von $o'.m(a)$ das Argument a vom (statischen) Typ A sein. Damit sind wiederum alle Argumente vom Typ $A'|A <: A'$ zulässig. Wird weiterhin der Ergebnistyp B zu B' (also $B' <: B$) spezialisiert, so kann $A \rightarrow B$ zu $A' \rightarrow B'$ spezialisiert werden.
Der Aufruf

```
InstanceTypeOf(c)(o').m(a)
```

wird somit zu

```
InstanceTypeOf(c')(InstanceTypeOf(c)(o')).m(a)
```

konvertiert.
$A <: A'$ muß auch dann gelten, wenn A eine Liste von Argumenten ist (siehe auch 2.1.2). Der zur Listenbildung verwendete Operator $\times$ heißt kovariant.

Definition 2.5 :
Ein Operator $\times$ mit $A \times B = (A, B)$ heißt *kovariant*, falls aus $A <: A'$ und $B <: B'$ die Subtypbeziehung $A \times B <: A' \times B'$ folgt.

Definition 2.6 :
Ein Operator $\rightarrow$ heißt *kontravariant (im linken Argument)*, falls $A \rightarrow B <: A' \rightarrow B'$ aus $A <: A'$ und $B' <: B$ folgt.

Eine zweite Form der Methodenspezialisierung wird implizit für ererbte Methoden ausgeführt. Hierfür werden Methoden zunächst als Funktionen, die an erster Stelle ein Argument vom Typ $InstanceTypeOf(classname)$ enthalten betrachtet.

$$pm_1 : (InstanceTypeOf(c) \times A_1) \to B_1 \tag{2.13}$$

Für pm_1 gilt wegen

$$InstanceTypeOf(c) \times A_1 <: InstanceTypeOf(c') \times A_1 \tag{2.14}$$

$$pm_1 <: (InstanceTypeOf(c') \times A_1) \to B_1 \tag{2.15}$$

Damit ist pm_1 durch Typreduktion auch auf $o' : InstanceTypeOf(c')$ anwendbar. pm_1 kann von c' geerbt werden. Insbesondere gilt auch

$$(InstanceTypeOf(c) \times A_1) \to B_1 <: (InstanceTypeOf(c') \times A_1') \to B_1' \tag{2.16}$$

für alle $A_1' <: A_1$ und $B_1 <: B_1'$. Im Gegensatz zum Überschreiben von Methoden werden also bei der Vererbung die Argumente spezialisiert, wohingegen die Funktionsrückgabewerte generalisiert werden.
Um die Möglichkeiten der Methodenspezialisierung voll zu nutzen, führt [AC96] den Typbezeichner "**Self**" ein. **Self** ist durch

$$\mathbf{Self} \triangleq InstanceTypeOf(c) \tag{2.17}$$

immer an eine Klasse c gebunden[5].

Beispiel:

```
class c is
    method m(): Self is ... end ;
end ;
subclass c' of c is
    method m(): Self is ... end ;
end ;
```

$TypeOf(o.m()) = InstanceTypeOf(c)$ und
$TypeOf(o'.m()) = InstanceTypeOf(c')$.

Um Definition und Implementation einer Klasse besser voneinander abgrenzen zu können, ist es sinnvoll, den zugehörigen *Objekttyp* dieser Klasse einzuführen.

```
class c is
    var attr : typeid ;
    method m(p:ArgType):RetType is ... end ;
end ;
```

[5] $x \triangleq y$ soll im Weiteren bedeuten: x wird durch y definiert.

```
ObjectType C is
    var attr : typeid ;
    method m(p:ArgType):RetType ;
end ;
```

Dieses Schema gestattet die separate Implementation der Klasse. Zusätzlich bietet es die Möglichkeit, die Implementation zu variieren, ohne das *Interface* zu verändern. Die Objekttypspezifikation wird auch *Objektprotokoll* genannt. Damit folgt zusammen mit 2.11

$$c' :: c \Rightarrow ObjectTypeOf(c') <: ObjectTypeOf(c) \tag{2.18}$$

Neben den bisher genannten Möglichkeiten zur Beschreibung von Klassen stellt die *Template*-Technik ein mächtiges Werkzeug zur Wiederverwendung von Code dar. Sie gestattet es, Code unter Einhaltung der geforderten Subtyprelationen auf unterschiedliche Basistypen anzuwenden.

Beispiel:

ObjectType CT **is**
    method m(x:A);
end ;
ObjectType DT **is**
    method m(x:B);
end ;
	ObjectOperator CTT[F <: A] **is**
	    method m(x:F);
	end ;
	ObjectOperator DTT[F <: B] **is**
	    method m(x:F);
	end ;

Wird für die Objekttypen in der linken Spalte $DT <: CT$ und $B <: A$ gefordert, so kann ein Objekt vom Typ DT nach CT reduziert werden. Damit kann auf eine Instanz vom Typ DT eine Methode $m : A \rightarrow$ angewandt werden. Dies führt sowohl bei statischer als auch bei dynamischer Bindung zu einem Konflikt bei der Argumentübergabe, wenn das Argument vom Typ $C_A B$ ist [6].
Dieses Problem wird durch die Parametrisierung nach einem Typ F (wie in der rechten Spalte angegeben) gelöst. Wird F durch einen vorgegebenen Typ (A, B) eingeschränkt, so spricht man von *eingeschränkter Typparametrisierung*[7]. Sei $B <: A$, so gilt im Beispiel $DTT[F] <: CTT[F]$ für alle $F <: B$.
Speziell die Vererbung von Methoden mit Argumenten vom Typ **Self** in kontravarianter Position induziert nicht notwendig eine Subtyprelation zwischen den Instanztypen der abgeleiteten und der Superklasse. Jedoch läßt sich für Objekttpypen T und S zu Klassen (sc::tc)

```
ObjectType T is ... end ;
ObjectType S is ... end ;
```

[6] $x : C_A B \Leftrightarrow B <: A \wedge x : A \wedge \neg x : B$

[7] Trellis/Owl, Sather, Eiffel, PolyTOIL und Rapide unterstützen eingeschränkte Parametrisierung. C++ unterstützt "nur" einfache Parametrisierung.

über die zugehörigen *Objektprotokolle* $T - Protocol$ und $S - Protocol$

ObjectOperator `T-Protocol [ X ]` **is** ... **end** ;
ObjectOperator `S-Protocol [ X ]` **is** ... **end** ;

eine *Subprotokollrelation* $< \sharp$ gemäß

$$S <: T - Protocol[S] \Rightarrow S < \sharp\, T \tag{2.19}$$

erklären. Die Subprotokollrelation impliziert nicht die Reduktionseigenschaft. Es folgt also nicht notwendig $s : T$ aus $s : S$ und $S < \sharp\, T$.
Aus der Subtyprelation höherer Ordnung

$$P \prec: P' \Leftrightarrow P[T] <: P'[T] \forall T \tag{2.20}$$

ergibt sich eine äquivalente Definition der Subprotokollrelation höherer Ordnung zu

$$S - Protocol \prec: T - Protocol \Rightarrow S < \sharp\, T \tag{2.21}$$

Objektbasierte Beschreibung

Im Unterschied zum klassenorientierten Ansatz wird in objektbasierten Sprachen durch die Definition

```
ObjectType C is
    var attr : typeid;
    method m ( ... );
end ;
object c:C is
    var attr : typeid;
    method m ( ... ) is ... end ;
end ;
```

grundsätzlich ein Objekt `c` generiert. Durch die Hinzunahme einer Methode `new` gemäß

```
procedure new (...) : C is
    object c : C is
        var attr : typeid;
        method m ( ... ) is ... end ;
    end ;
    return c;
end ;
```

zur Objektdefinition ist die Bildung von Objektsammlungen möglich.
Einen anderen Zugang findet man im Prototypenkonzept. Hier werden zunächst vollständige Instanzen von Klassen erzeugt, von denen einige später als "kanonische Repräsentanten" oder *Prototypen* interpretiert werden können.
Neue Instanzen werden durch *Klonen* erzeugt.

```
var cc:C := clone protoC;
```

Beim Klonen wird eine "flache" Kopie[8] des Argumentes angelegt (Abbildung 2.4). Erst durch *Mutation* (das ist das dynamische Verändern von Objekteigenschafte) ist eine Spezialisierung von Objekten möglich. Das Überschreiben von Methoden wird in objektbasierten Sprachen durch das Konzept der Methodenneudefinition (*Update*) ersetzt.

```
cc.m := method (...) is ... end ;
```

Prinzipiell können Klonen als Vererbung beziehungsweise Update als Überschreiben im Klassenkonzept angesehen werden. Allerdings fehlt beim simplen

[8] Flaches Kopieren überträgt Referenzen auf die Komponenten der rechten Seite. Die Kopie hat aber einen unabhängigen Zustand.

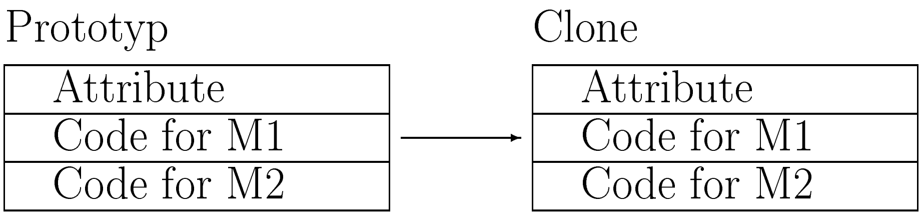


Abbildung 2.4: Schematische Darstellung des flachen Kopierens

Klonen noch die Möglichkeit, Objekte zu erweitern. Ein Erweiterungsmechanismus wird durch einen *Spender-Empfänger* Ansatz verfügbar gemacht. Die Attribute des Spenders können explizit oder implizit übernommen werden. Eine andere Klassifikation beschreibt das Einbinden von Attribute in den Empfänger (*embedding*) oder deren Delegation an den Spender (*delegation*).
Objektbasierte Sprachen unterstützen diese Verfahren durch entsprechende Konstrukte.

Beispiel: *embedding*-Paradigma

```
object c:C is
    var a1 : typeid;
    method m is ... end ;
end ;
```

explizite Weitergabe

```
object cc:C' is
    var a1 : typeid := c.a1;
    var a2 : typeid ;
    method m is
        embed c.m
    end ;
end ;
```

implizite Weitergabe

```
object cc:C' extends c is
    var a2 : typeid ;
    override m is
        embed c.m
    end ;
end ;
```

Der Code einer durch Einbinden weitergegebenen Methode wird durch das **embed** Konstrukt in das neue Objekt kopiert.
Im Delegationsansatz erhält das abgeleitete Objekt einen Verweis auf das Spenderobjekt und kann mit Hilfe eines **delegate** Konstruktes auf den Methodencode des Spenders zugreifen. Der Spendercode wird nicht in das neue Objekt kopiert.
Der objektbasierte Ansatz bietet als Besonderheit die Möglichkeit der *dynamischen Vererbung* im Delegations-Paradigma. Wird der Verweis auf das Spenderobjekt geändert, so spricht man von dynamischer Vererbung (Abbildung 2.5).
Vergleichbar zu Methodensammlungen im klassenbasierten Ansatz gibt es im objektbasierten Schema sogenannte *trait*-Objekte. Traits sind Prozedursamm-

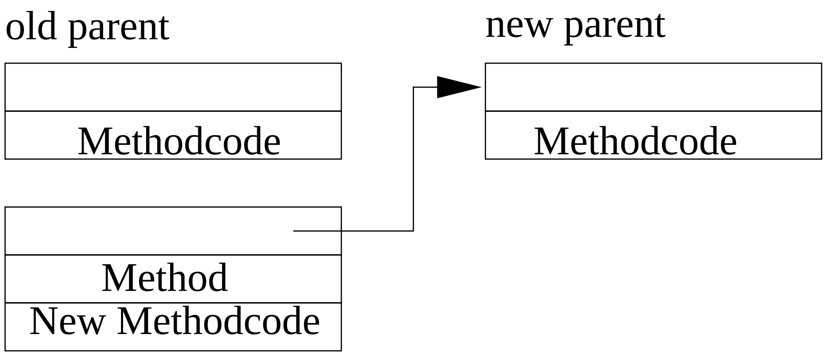


Abbildung 2.5: Prinzip der dynamischen Vererbung

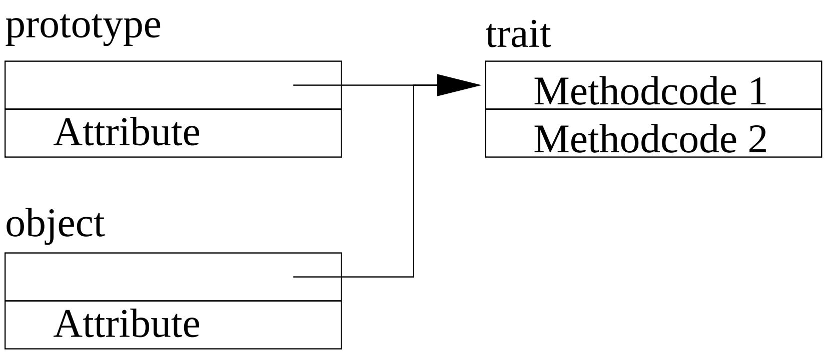


Abbildung 2.6: Klassenbildung durch Prototypen und Traits

lungen. Prototypen enthalten Verweise auf zugehörige Traits, die beim Klonen an die Kinder weitergegeben werden. Verweise auf Traits können prinzipiell auch geändert werden (Abbildung 2.6).

2.2.2 Theorie der Objektorientierten Programmierung

In [AC96] wird, basierend auf dem ς-Kalkül eine Theorie des objektorientierten Programierens aufgebaut.
[AC96] erklrt ein Objekt als Tupel aus ς-Komponenenten.

$$\mathcal{O} \triangleq [l_1 \triangleq \varsigma(p_1, ..., p_n)T(p_1, ..., p_n), ...] \tag{2.22}$$

ς bindet beim Aufruf einer Methode l_k den Parameter **self** an die Methode. Dabei wird nicht zwischen Attributen und Methoden unterschieden. Das Beschreiben eines Attributes wird durch das Konzept des Methoden-update bzw. die Abfrage eines Attributwertes durch den Aufruf einer Methode ersetzt.

Eine Klasse besteht nach [AC96] aus einer Sammlung von *Pre-Methoden* (*Trait*) und einem Konstruktor. Das ist eine Funktion im *lambda*-Kalül, die den ς-Parameter einführt und als Ergebnis ein Objekt vom Typ "Objekt dieser Klasse" liefert.

Kapitel 3

OO-Sprachen

Inhalt

Turbo-Pascal gilt als die erste OO Sprache im PC Bereich. Wir stellen die Sprache vor und erläutern die Besonderheiten im Umgang mit OO Paradigmen. Wir erläutern außerdem die Implementation der *virtual method table* [Su93]. In den weiteren Abschnitten fahren wir in der gleichen Art und Weise mit die Sprachen Oberon, Smalltalk-80, C++, Java, Ada 95 und Eiffel. D.h. wir geben immer eine kurze Einführung in die Sprache und erläutern dann die OO Spezifika.

3.1 Turbo-Pascal

Nils Schmeißer, M. Zouhir Barakat

Pascal wurde 1970 von N. Wirth entworfen ([JW85]) und 1983 standardisiert (ISO 7185, DIN 66256). Pascal existiert mittlerweile in verschiedensten Dialekten mit vielen nützlichen Erweiterungen (UCSD-PASCAL, Microsoft Pascal, PASCAL MT+ und Turbo1-Pascal). Letzteres von Borland, welche 1989 mit der Einführung der Version 5.5 den Schritt zum objektorientierten Programmieren durchführten (Turbo-Pascal V 5.5 war die erste OO-Sprache im PC-Bereich, noch vor C++). Turbo-Pascal ist bisher auch der einzige Pascal Dialekt, der objektorientierte Programmierung zuläßt.
Pascal gilt als die erste, klar strukturierte Programmiersprache, die außerdem ein sehr scharfes Typkonzept verfolgt, aber noch keine modulare Programmierung kannte (das Unit-Konzept ist eine Erweiterung (UCSD, Borland)!). [Heu92] ordnet Turbo-Pascal den *Hybridsprachen* zu, d.h. Erweiterungen imperativer und funktionaler Programmiersprachen.
Wir werden uns hier auf Turbo-Pascal ab Version 5.5 beschränken (z.Zt. aktuell ist 7.0). Turbo-Pascal (TP) lehnt sich von der Syntax her natürlich an Pas-

```
type classname = object
          public_components;
          virtual_method; virtual ;
          constructor constr ( args ); { Konstruktor }
          destructor destr ( args ); virtual; { Destruktor }
     private:
          private_components;
end ;
```

Abbildung 3.1: Klassendefinition

cal an, wobei das Standardkonstrukt zur Einleitung des Quelltextes (**program name(in,out);**) entfallen kann. Als Erweiterung kennt Turbo-Pascal das Unit-Konzept (Modulkonzept). Ein Modul entspricht in TP immer genau einer Unit. Eine Unit ist immer eine Datei und besteht aus dem Spezifikator **unit name;** optional gefolgt von einem **interface** Teil (hier werden die exportierbaren Identifikatoren deklariert) und einem **implementation** Teil (Definitionsteil). Eine Unit kann einen Hauptblock im Implementationsteil enthalten, der vor Eintritt in den Hauptblock des Programmes abgearbeitet wird. Units werden mit der **use** Anweisung importiert (eigentlich werden nur die Deklarationen importiert, die übersetzten Module werden vom Linker zusammengebunden).
Für eine genaue Beschreibung der Syntax verweisen wir hier auf [Su93] oder Handbücher ab Version 5.5.

Klassendefinition

Ein Klasse wird in TP als strukturierter Datentyp (Record) aufgefaßt (Abb. 3.1), der im Typvereinbarungsteil deklariert wird. Das geht konform mit dem Konzept des abstrakten Datentyps (ADT).
Die Eigenschaft, ein Objekt zu sein, wird durch die Verwendung des Schlüsselwortes **object** an Stelle von **record** ausgedrückt. Komponenten dieser Struktur können neben typisierten Elementen auch Prozeduren und Funktionen sein (wir werden im Abschnitt 3.2 sehen, daß dies durchaus nicht grundsätzlich so sein muß). Für die Komponenten kann man Zugriffsspezifikationen angeben (**private** und **public**), d.h. auf eine, als privat deklarierte Komponente kann nur innerhalb der Instanz zugegriffen werden, öffentlich Komponenten sind von überall her sichtbar. TP sieht die Einrichtung einer Klasse in zwei Abschnitten vor; a) Deklaration der Klasse b) Definition der Methoden. b) findet außerhalb des Typvereinbarungsteils statt. TP verwendet zur Identifikation eine Qualifikation *Klassennname* **.** *Komponentenname*, die Definition einer Methode würde also wie folgt aussehen:

```
procedure classname . method ( args );
     body
function classname . method ( args ) : returntype;
     body
```

Innerhalb einer Methodendefinition kann ohne Qualifikation auf alle Komponenten der Klasse zugegriffen werden. TP kennt zwei spezielle Methode, den Konstruktor (**constructor**) und den Destruktor (**destructor**). Dem Destruktor kommt eine besondere Bedeutung bei der Verwendung dynamisch erzeugter Instanzen zu. Selbst wenn der Körper des Destruktors leer ist, hängt der Compiler Code an, der die Aufgaben einer garbage-collection übernimmt, d.h. der im Heap (Freispeicher zur Ablage dynamisch erzeugter Daten) belegte Platz wird freigegeben. Borland empfiehlt den Destruktor grundsätzlich als virtuelle Methode zu deklarieren.

Vererbung

TP kennt nur einfache Vererbung. Die Oberklasse wird in "**()**" hinter dem Schlüsselwort **object** angehangen.

type classname = **object** **(** parent **)**;

In der Definition kann der Zugriff auf Komponenten der Oberklasse durch die Qualifikation (*Oberklasse* **.** *Komponente*) erfolgen. Der Zugriff ist nur auf nicht-private Komponenenten möglich[1].

Instanzierung, Zugriff und Destruktion

T.-Pascal kennt neben statischen Instanzen auch dynamische. Erstere werden bei Eintritt in ihren Gültigkeitsbereich erzeugt und bei Verlassen zerstört, dynamische werden erst durch den Ruf an **new** erzeugt. Jede Klasse, die eine virtuelle Methode hat, muß einen Konstrukor haben und dieser muß **vor** dem Zugriff auf eine virtuelle Methode gerufen werden.
TP kennt einen speziellen Parameter, der implizit als letztes Argument an Methoden weitergeleitet wird. Der **self**-Parameter (Variablenparameter) enthält einen Verweis auf die, die Methode rufende Instanz.
Borland hat die Syntax und Semantik von **new** erweitert und etwas verändert (!Funktion), so daß die Erzeugung einer dynamischen Instanz wie in Abb. 3.2 erfolgen kann. **new** verlangt dazu einen Typbezeichner *Zeiger auf Klasse*, um die Klassenzugehörigkeit feststellen zu können (das darf kein "^typid" Konstrukt sein, sondern muß ein Typidentifikator sein).

[1] [Heu92] gibt eine Quelle an, nach der in Version 6.0 **private** nicht richtig funktioniert und der Zugriff auch auf private Komponenten möglich ist.

- statisch: **var** instvar **:** classname;
  instvar **.** constr **(** args **)**;
  instvar **.** component;
  instvar **.** destr;

- dynamisch: **var** instref **:**ˆclassname;
  instref **:=** **new** **(** pclassname **,** constr **(** args **)** **)**;
  instrefˆ**.** component;
  dispose **(** instref **,** destr **(** args **)** **)**;

Abbildung 3.2: statische und dynamische Instanzen und Zugriff

Kommentar

Borland's Turbo-Pascal war die erste Sprache im PC Bereich, mit der objektorierntiertes Programmieren möglich war (Turbo Pascal V 5.5 , noch vor diversen C++ Implementationen (!)). Pascal bietet von Haus aus ein sehr strenges Typkonzept, das sogar noch schärfer als in C++ arbeitet (TYPE banane=integer; pflaume=integer; $\Rightarrow$ banane $\neq$ pflaume).
Turbo-Pascal kennt keine Mehrfachvererbung, keine Operator- und Funktionsüberladung und kann keine strukturierten Datentypen als Funktionsrückgabewert enthalten. Damit ist man auf die Verwendug von Referenzen angewiesen, was auf Grund einer nicht vorhandenen garbage-collection mitunter Probleme bereiten kann.
Die Zuweisung statischer Instanzen erfolgt unter Wertsemantik.

3.2 Oberon

Andreas Knüpfer

"So einfach wie möglich, aber nicht einfacher" (Albert Einstein)

3.2.1 Überblick

Oberon wurde von Niklaus Wirth und Jürg Gutknecht ab 1985 entwickelt, die sich vorgenommen hatten, ein leistungsfähiges und bequemes sowie überzeugend erklär- bares Betriebssystem für Arbeitsplatzrechner zu entwickeln. Um nicht von einschränk- enden Randbedingungen behindert zu werden, entschlossen sie sich zu einem völligen Neuanfang. Der Name des Projektes stammt von dem Uranus-Mond Oberon, der 1985 gerade von einer der Voyager-Sonden passiert wurde. Ziel war es, ein kleines System zu schaffen, das relativ wenige, jedoch fundamentale Konzepte beinhaltet und effizient implementierbar ist. Die selben Merkmale gelten für die Programmiersprache Oberon, die nur einen Teil dieses Projektes darstellt. 1991 erweiterten Niklaus Wirth und Hanspeter Mössenböck die Sprache um einige wenige aber nützliche Neuerungen. Oberon-2 blieb dabei voll aufwärtskompatibel. Im Folgenden wird nur Oberon-2 beschrieben, Erweiterungen werden lediglich erwähnt.
Oberon reiht sich ein in eine Familie von Programmiersprachen, die begründet wurde von Algol60.
Algol60 (1960) ist eine der ersten Programmiersprachen, die allgemein und nicht auf einen speziellen Rechner zugeschnitten war. Die Programme sollten nicht nur von einem Rechner ausgeführt werden können, sondern auch für Programmierer leicht lesbar und verständlich sein.
Neu in Pascal (1970), dem wahrscheinlich erfolgreichste Nachfolger von Algol60 waren u.a. Mengendefinitionen, Records, also neuen Datentypen, strukturierten Dateien sowie Zeiger, was dynamische Datenstrukturen ermöglicht.
Modula-2 (1979) führte das Modulkonzept ein, eingeschlossen dem Geheimnisprinzip und getrennterDCbersetzung. Es beherrscht offene Arrays und unterstützt speziell maschinennahe Programmierung.
Die wichtigste Neuerung in Oberon (1988) und Oberon-2 (1991) waren das Konzept der Typerweiterung und der typgebundenen Prozeduren. Des weiteren verschwanden eine Reihe von Merkmalen, die Modula-2 noch unterstützte, um Oberon - Programme systemunabhänig zu machen.

3.2.2 Die Sprache Oberon-2

Die Sprache Oberon ist nicht auf Oberon-Systeme beschränkt. Es existieren auch ganz normale Compiler für gängige Betriebssysteme. Ihre Hauptmerkmale sind:

- Blockstruktur,
- Modulkonzept,
- getrennte Übersetzung,
- strenge Typprüfung,
- Typerweiterung und
- typgebundene Prozeduren (allerdings erst in Oberon-2)

Syntax und Semantik von Oberon sind sehr ähnlich und teilweise identisch zu Modula-2 oder Pascal. Allerdings werden Groß- und Kleinschreibung unterschieden und alle Terminalsymbole müssen in Großbuchstaben erscheinen.

vordeklarierte Datentypen

Oberon verfügt über eine Basismenge vordefinierter Datentypen, der Wertebereich mitunter implementationsabhängig etwas abweichen kann.

	Name	typischer Wertebereich
Ganze Zahlen	SHORTINT	-128 .. 127
	INTEGER	-32768 .. 32767
	LONGINT	-2147483648 .. 2147483647
Gleitkommazahlen	REAL	+/- 3.40282 E 38 (32 Bit)
	LONGREAL	+/- 1.79769 D 308 (64 Bit)
ASCII-Zeichen	CHAR	0X .. 0FFX (hexadezimal)
Boolescher Typ	BOOLEAN	TRUE, FALSE
Mengen	SET	0 .. 31

Typdeklarationen

Array :

```
Vektor: ARRAY n OF REAL;
Matrix: ARRAY m, n OF REAL;
Parameter: ARRAY OF INTEGER;,
```

nur als formaler Parameter erlaubt
Arrays werden durch ganze Zahlen indiziert, beginnend bei 0, z.B.:
`Vektor[0]`, `Matrix[i,j]`

Record :

```
Person 3DRECORD
Name, Vorname: ARRAY 16 OF CHAR;
Alter: SHORTINT;
...
END;
```

Felder von Records werden angesprochen über Variablenname.Feldname, z.B.: `Person.Name`, `Person.Alter`

Zeiger :

`PersonPtr3D POINTER TO Person;`
`Box3D POINTER TO RECORD x, y, b, h: INTEGER END;`
`String3D POINTER TO ARRAY OF CHAR;`, offenes Array erlaubt als Zeigerbasistyp

Ist `p` vom Typ `PersonPtr`, kann man mit `NEW(p)` eine neue Instanz des Typs `Person` erzeugen. `p` enthält dann eine Referenz auf diese Instanz. Sie wird über `p^.Name` oder einfach `p.Name` angesprochen.
Achtung: `q:=3Dp` kopiert im Gegensatz zu `q^:=3Dp^` nur flach! Ist `s` vom Typ `String`, so erzeugt `NEW(s,n)` eine dynamische Variable vom Typ `String` mit der Länge `n`, ansprechbar als s^.[i] oder s[i].
Es existiert keine explizite Freigabe von dynamischen Variablen bzw. Datenstrukturen. Vielmehr sorgt die automatische Garbage Collection dafür, daß nicht mehr referenzierte Speicherobjekte freigegeben werden. Zeiger, die auf kein Ziel zeigen, enthalten den Wert `NIL`.

Prozedurtypen

Prozedurvariablen enthalten eine Referenz auf eine Prozedur (oder NIL). Wird die Prozedurvariable als Prozedur aufgerufen, so wird die "wirkliche" Prozedur ausgeführt, auf die die Variable gerade verweist. Siehe Beispiel:

```
VAR Write: PROCEDURE(ch:CHAR);

PROCEDURE WriteBildschirm(ch:CHAR);
BEGIN
...
END WriteBildschirm;

Write:=3D WriteBildschirm;
```

Prozedur-Deklaration entsprechend dem Beispiel:

```
PROCEDURE Add(a, b: INTEGER):INTEGER;
```

Ein Rückgabetyp kann optional angegeben werden. Dann handelt es sich um eine Funktion.

```
VAR s: INTEGER;

 (* an dieser Stelle ist die Deklaration lokaler Prozeduren
    moeglich *)

BEGIN
 s:=3D a + b;
 RETURN s      (* gibt den Wert zur"uck, wenn es sich um eine
                  Funktion handelt *)
END Add;
```

Rekursive Aufrufe sind möglich, lokale Variablen werden dabei in jedem Rekursionsschritt neu angelegt (die Übergabe von Parametern und Anlage statischer Instanzen lokaler Variablen erfolgt auf dem Stack).

Kontrollstrukturen

Oberon kennt die nachfolgend aufgeführten Kontrollstrukturen:

While-Schleife :
`WHILE` Bedingung `DO` Anweisungen `END`;

Repeat-Schleife :
`REPEAT` Anweisungen ; `UNTIL` Abbruchbedingung;

Loop-Schleife :
`LOOP` Anweisungen; `EXIT;` Anweisungen; `END`;

For-Schleife : (neu in Oberon-2)
`FOR i:=3D a TO e DO` Anweisungen; `END`;

IF-Anweisung :
`IF` Bedingung `THEN` Anweisungen ; `ELSIF` Bedingung `THEN` Anweisungen; ... `ELSE` Anweisungen; `END`;

Case-Anweisung :
`CASE` Variable `OF` werteliste1: Anweisungen; werteliste2: Anweisungen; ... `ELSE` Anweisungen; `END`;

Module

Der Aufbau von Modulen soll am folgenden Beispiel beschrieben werden:

```
MODULE Rechner;
IMPORT In, Out;         (* Die Module In und Out werden
                           importiert *)
```

```
VAR a, b: INTEGER;  (* Deklaration der globalen Variablen *)

PROCEDURE Add*;         (* Der Stern markiert die Prozedur
                           als exportierbar *)
BEGIN
  Out.Int(a+b,5);       (* Prozeduren anderer Module werden mit
                           Modulname.Prozedurname aufgerufen *)
  Out.Ln;
END Add;

PROCEDURE Sub*;
BEGIN
  Out.Int(a-b,5);
  Out.Ln;
END Sub;

PROCEDURE Mul*;
BEGIN
  Out.Int(a*b,5);
  Out.Ln;
END Mul;

PROCEDURE Div*;
BEGIN
  IF (b#0) THEN
    Out.Int(a DIV b,5);
  ELSE
    Out.String("Division durch Null");
  END;
  Out.Ln;
END Div;


PROCEDURE Print*();
BEGIN
  Out.Open;
  Out.String("a=3D");
  Out.Int(a,5);
  Out.String(", b=3D");
  Out.Int(b,5);
  Out.Ln;
END Print;
```

```
PROCEDURE A*();
BEGIN
  In.Open;
  In.Int(a);
  IF (In.Done) THEN
    Print;
  ELSE
    Out.String("Fehler");
  END;
END A;

PROCEDURE B*();
BEGIN
  In.Open;
  In.Int(b);
  IF (In.Done) THEN
    Print;
  ELSE
    Out.String("Fehler");
  END;
END B;

BEGIN
  a:=3D0;
  b:=3D0;
END Rechner.
```

Das Exportmarke `*` direkt hinter einem Prozedur- oder Variablennamen in der Deklaration macht die betreffende Variable oder Prozedur nach außen sichtbar. D.h. Module, die dieses Modul importieren, können auf diese Variablen zugreifen bzw. die Prozeduren aufrufen. Für Variablen gibt es eine weitere Abstufung: Die Exportmarke `-` nach dem Variablennamen, macht die Variable für importierende Module zwar sichtbar, verleiht ihr aber Nur-Lese-Status.
Der Block zwischen dem letzten `BEGIN` und dem abschließenden `END` ist für die Initialisierung des Moduls direkt nach dem Laden zuständig. Wie man am begrenzten Umfang sehen kann, ist das nicht das eigentliche Hauptprogramm. Wo befindet sich dann das Hauptprogramm?
In Oberon gibt es keinen festen Programmeinstiegspunkt, wie üblich. Tatsächlich stellt ein fester Programmeinstiegspunkt eine Einschränkung dar. Oberon läßt jede exportierte Prozedur eines Moduls als Einstiegspunkt zu, die keine Parameter hat. Diese Prozeduren heißen auch Kommandos. Da ein Modul in der Regel (, wenn nicht explizit entladen) nur einmal geladen wird, danach im Speicher bleibt und seinen Zustand behält, kann man über seine Kommandos mit einem Modul kommunizieren. Das folgende Beispiel demonstriert das:

`Rechner.A 100`
: Es wird zum ersten Mal ein Kommando des Moduls Rechner aufgerufen. Also wird das Modul geladen, die Initialisierung und anschließend das Kommando werden ausgeführt. Der folgende Parameter ist kein formaler Parameter der Prozedur, sondern kann als Text von der Prozedur interpretiert werden (ähnlich einem MS-DOS Befehlszeilen-Parameter).

`a=3D 100, b=3D 0`
: die Ausgabe des Kommandos

`Rechner.B 20`

`a=3D 100, b=3D 20`
: Ein weiteres Kommando wird aufgerufen. Das im Speicher befindliche Modul führt es direkt aus. Man beachte, daß der Wert von a zwischen den Aufrufen erhalten blieb.

`Rechner.Add 120`
: Das Kommando Add verlangt keine Eingabe sondern greift auf die vorher eingegebenen Werte zurück, berechnet deren Summe und gibt sie aus.

`Rechner.Sub 80`

`Rechner.Mul 2000`

`Rechner.Div 5`
: wie vorher

`Rechner.Print`

`a=3D 100, b=3D 20`
: Das Kommando Print gibt den Zustand des Moduls aus.

`System.Free`

`Rechner ~System.Free`
: Rechner unloading, Hier wird das Modul Rechner explizit aus dem Speicher entfernt (was nur möglich ist, wenn das Modul von keinem anderen aktiven Modul importiert wird).

`Rechner.Print`

`a=3D 0, b=3D 0`
: Durch das Kommando Print wird das Modul Rechner erneut geladen und neu initialisiert. Der vorherige Zustand des Moduls ist verlorengegangen.

Klassen und Objekte

Klassen werden in Oberon als Erweiterung von Records behandelt.

```
TYPE AClassPtr=POINTER TO ClassA;
     ClassA=RECORD
                private_method : PROCEDURE prochead1;
                public_method *: PROCEDURE prochead2;
                private_attr   : typ1;
                public_attr   *: typ2;
            END;
```

Vererbung bzw. Typerweiterung wird ausgedrückt wie folgt:

```
TYPE BClassPtr=POINTER TO BClass;
     BClass=RECORD(StackDesc)
                new_method     : PROCEDURE prochead;
            END;
```

`IntStackDesc` ist also von `StackDesc abgeleitet`. Es enthält neben den neu definierten Datenfeldern auch die der Elternklasse. Ebenso deren Methoden sofern die Methoden in der abgeleiteten Klasse nicht überschrieben wurden. Die Typen `StackDesc` und `IntStackDesc` sind kompatibel. D.h. Eine Variable vom Typ `IntStackDesc` kann einer `StackDesc`-Variablen zugewiesen werden, nicht aber umgekehrt, da sonst die neu hinzugekommenen Datenfelder undefiniert blieben.
Methoden werden an eine Klasse gebunden, indem die Klasse als Typ eines besonderen formalen Parameters vor dem Prozedur erscheint:

```
PROCEDURE (s:IntStackPtr) TOS*():LONGINT;
BEGIN
  RETURN s.i;
END;
```

Ursprünglich wurden in Oberon Methoden nicht an Klassen, sondern an Objekte gebunden. Dabei enthielten die Objekte Prozedur-Variablen als Felder, die explizit zugewiesen werden mußten. Diese Vorgehensweise kam zwar ohne ein zusätzliches Sprachkonstrukt aus, hatte aber den Nachteil, das Speicherplatz in den Objekten verschwendet wurde und das Objekte einer Klasse untereinander inkonsistent bzgl. ihrer Methoden sein konnten.
Die Datenfelder der Klasse bzw. eines Objektes der Klasse für die Methoden zugreifbar. Allerdings werden diese Datenfelder nicht wie z.B. in Turbo Pascal direkt mit ihrem Namen angesprochen sondern über den formalen Parameter, der das aktuelle Objekt bezeichnet.
Wird eine Methode der Elternklasse von einer abgeleiteten Klasse überschrieben, muß die Schnittstelle identisch bleiben. Neue Methoden können hinzugefügt werden.

Ein neues Objekt einer Klasse wird mit `NEW` angelegt, z.B.:

```
VAR stapel:IntStackPtr;
...
NEW(stapel);
```

Oberon sieht keine Konstruktoren vor. Die Initialisierung eines Objektes sollte über eine normale Prozedur erfolgen, in der das Objekt als VAR-Parameter erscheint. Es ist nicht ratsam die Initialisierung als Methode zu implementieren, da keine Möglichkeit besteht, neue Datenfelder als neue formale Parameter der Schnittstelle hinzuzufügen. Ebenso sollte die `NEW`-Anweisung nicht in der Initialisierungsprozedur enthalten sein. Dann wäre es nicht mehr möglich, in der Initialisierung eines Objektes einer Klasse auf die Initialisierungsroutine der Elternklasse zurückzugreifen.
Das explizite Zerstören von Objekten ist nicht möglich. Die Garbage Collection sorgt wiederum dafür, nicht mehr referenzierte Objekte zu löschen.
Anmerkung (d. Ed.):

3.2.3 Das Oberon - System

Einer der Hauptkritikpunkte an existierenden Systemen war, daß durch die "gewöhnlichen" Programmierpraxen vor allem in großen Systemen gleiche oder ähnliche Routinen vielfach vorhanden sind und nicht wiederverwendet werden (können).
Mit ihrem neuen System wollen N. Wirth und J. Gutknecht dem Trend umkehren, daß die Software mit ihrem zunehmenden und nicht immer gerechtfertigten Ressourcen- verbrauch den Fortschritt der Hardwareentwicklung wieder aufhebt. Das Oberon-System demonstriert in beeindruckender Art und Weise, wie ein modernes und benutzerfreundliches , komplettes System implementiert sein kann. So umfaßt der Objektcode des sog. äußeren Kerns des ursprünglichen Oberon - Systems, bestehend aus dem Kernel-Modul, dem dynamischen Modullader, der Garbage Collection, dem Dateisystem, den primären Treibern, LAN-Unterstützung, dem Fenstersystem, einem Texteditor und dem Compiler, weniger als 200 KB . Dabei sind alle enthaltenen Module bis auf das Kernel- und das Display-Modul auch in der Sprache Oberon geschrieben.
In einem Oberon - System ist die Grenze zwischen Betriebssystem und Anwendungsprogramm fließend. Ein Programm fügt dem gesamten System eine Reihe von neuen Modulen hinzu sowie Kommandos, über die das Programm ansprechbar ist.

3.2.4 Hinweise zum Umgang mit Oberon

Neben der ursprünglichen Implementierung existieren eine Reihe von Portierungen auf andere Plattformen, teilweise auf andere Betriebssysteme aufsetzend und mit Erweiterungen, beispielsweise Farbgrafik oder die Unterstützung

hierarchischer Dateisysteme. Das Oberon System 3 enthält bereits ausführliche Dokumentationen und Tutorials. Weitere Informationen sind u.a. auf der Oberon-Homepage [Obe] verfügbar.
Oberon arbeitet mit einer Drei-Tasten-Maus, die zwar per Tastatur emuliert werden kann, eine echte Drei-Tasten-Maus ist aber dringend zu empfehlen.
Kommandos werden aufgerufen indem man Modulname.Kommando , evtl. auch Parameter, in ein Textfenster schreibt und mit der mittleren Maustaste darauf klickt. Zum Beispiel in den Menüleisten der Fenster sind einige Kommandos bereits aufgeführt, so daß man nur noch darauf klicken muß. Außerdem existieren auch Buttons und Pull-Down-Menüs.
Ein ˆ hinter einem Kommando bewirkt, daß der zuletzt (mit der rechten Maustaste) selektierte Text als Parameter gelesen wird. Ein ∗ an dieser Stelle übergibt den Inhalt des gerade markierten Fensters als Parameter. Ein Fenster wird markiert, indem man `F1` drückt, während sich der Mauszeiger über dem Fenster befindet.
Eine neue Textdatei, z.B. ein Modul-Quelltext, wird erzeugt, indem man eine Datei mit dem gewünschten Namen mit `Edit.Open` ˆ oder dem `Open` ˆ - Button öffnet. Speichern und Schließen sind über die Menüleiste verfügbar. Übersetzt wird ein markierter Quelltext mit `Compiler.Compile` ∗ oder `Builder.Compile`. `Builder.MarkErrors` ˆ markiert vom Compiler gemeldete Fehler im Quelltext, wenn die Fehlerliste (im Fenster System.Log) selektiert ist. `Browser.ShowDef` oder `Watson.ShowDef` gefolgt von einem Modulnamen liefert die Schnittstelle des Moduls. Ein weiteres wichtiges Kommando beim Programmieren ist
`System.Free` Modulname˜. Damit wird ein Modul explizit aus dem Speicher entfernt, wenn kein anderes Modul dieses importiert. Das ist wichtig, wenn eine neue Version des Moduls verwendet werden soll. Andernfalls wird die alte Version weiter verwendet obwohl die neue erfolgreich kompiliert wurde.
`System.Quit` schließt das System.

3.2.5 Erfolg/Ausblick

Als Ausgründung aus dem Swiss Federal Institute of Technology entstand 1993 die kommerzielle Firma Oberon microsystems, Inc. . Diese Firma prägte die Bezeichnungen Oberon/F für das Framework, also das Betriebssystem Oberon sowie Oberon/L für die Sprache Oberon.
Inzwischen nennt sich Oberon/L Component Pascal. Damit will man eine Verbindung zu der über 30-jährigen Geschichte von Pascal herstellen: "Component Pascal steht dem ursprünglichen Pascal näher als das heutige Fortran seinem Orginal" [].
Außerdem will man auf die Komponenten-Orientierung hinweisen, d.h.
Erweiterbarkeits- und Sicherheitsaspekte betonen. Compiler für Component Pascal akzeptieren Oberon-2-Programme und unterstützen zusätzlich 64-bit Integer- und 2-Byte-Zeichen-Typen.

Oberon/F wurde ebenfalls umbenannt nach BlackBox Component Builder, die Klassenbibliothek nennt sich BlackBox Component Framework. "Black Box" soll zum Ausdruck bringen, daß der Inhalt der Komponenten völlig unbekannt bzw. egal ist, und nur die Schnittstelle eine Rolle für den Client-Programmierer spielt.
Auch wenn sie sich aus Akzeptanzgründen "wieder" Pascal nennt, hat/hätte Oberon ein sehr großes Potential. Die Sprache selbst ist übersichtlich, schlank und nicht zuletzt wegen seiner Pascal-Ähnlichkeit leicht und sofort verständlich. Einige Merkmale, die heute mit Java assoziiert werden, enthielt Oberon schon von Anfang an. So funktionieren kompilierte Module plattformübergreifend! Ebenso können Module während der Laufzeit hinzugefügt werden.
Abgesehen von der kommerziellen Nutzung ist Oberon hervorragend für Lehr-Zwecke geeignet. So bemerken die Autoren in [RW94] treffend: "Da Programmieren ... schwierig ist, müssen Studenten davor bewahrt werden, die zusätzliche Bürde einer komplexen Sprache zu tragen."

3.3 Smalltalk

Dominik Kostanjšek

3.3.1 Die Geschichte von Smalltalk und Smalltalk's Big Ideas

Smalltalk als Vision

- Entwicklungsbeginn Anfang der siebziger Jahre am „Xerox Palo Alto Research Center“
- Eine „Learning Research Group“ stellte sich die Aufgabe, wie ein Computersystem (Hard- und Software) auszusehen hat, mit dem der Benutzer effektiv und komfortabel arbeiten kann
- Im Jahre 1981 änderte diese Gruppe ihren Namen in „Software Concepts Group“ (SCG) um
- Ihr Ziel war ein Informationssystem, das sich den wandelnden Bedürfnissen seines Benutzers anzupassen vermag (Manipulation), das aber auch in seinem Können entwickelbar ist (Evolution)
- Zur Realisierung dieser Ziele konzentrierte man sich auf die Lösung zweier Problembereiche:
  - Eine **Problembeschreibungssprache** als eine Schnittstelle zwischen Gedankenmodellen und ihren Realisierungen auf einem Computer
  - Eine **Sprache zur Interaktion**, welche die menschliche Art zu kommunizieren, auf die Mensch-Maschine-Kommunikation zu übertragen versucht

wenige Konzepte

Der Kern des Smalltalk-Systems basiert auf fünf Konzepten:

- Objekt
- Nachricht
- Klasse
- Instanz
- Methode

Sie sind relativ schnell zu verstehen. Die eigentliche Mühe besteht darin, zu sehen, in welch verschiedenen Weisen sie im System angewandt werden.

Interaktive Programmierungebung

- Das System realisierte neue Gestaltungsprinzipien:
  - Eine Dialogsteuerung mit Maus und Fenstern
  - Eine Visualisierung der einzelnen Komponenten
  - Desktop-Paradigma
- Dazu wurde das System auf eigenständigen Rechnern mit Maus und hochauflösendem Bildschirm implementiert
- Integrierte Programmentwicklung soll die Entwicklung, Verwaltung und Überprüfung innerhalb des Systems und ohne es zu unterbrechen ermöglichen

Big System

- Smalltalk besteht aus einer Vielzahl von Komponenten, die es in die Nähe eines Betriebssystems kommen lassen:
  - automatische Speicherverwaltung
  - Dateisystem
  - display handling
  - Text- und Bildbearbeitung
  - Geräteverwaltung
  - Debugger
  - Scheduling
  - Compilation/Decompilation
  - performance spy
- Das Objekt-Nachrichten-Konzept unterliegt jeder Komponente und jeder Ebene des Systems
- Die Reduzierung der Komplexität wird erreicht durch:
  - Kapselung und definierte Informationsschnittstellen
  - Gruppierung ähnlicher Komponenten (Klassenkonzept)
- Benutzer-Klassen gleichberechtigt neben System-Klassen
- Vermeidung von Redundanz durch Bildung von Unterklassen (Vererbung)
- Prototyping wird ermöglicht

3.3.2 Objekte und Nachrichten

- Objekte repräsentieren Komponenten des Systems. Das können z. B. sein:
  - Zahlen
  - Zeichen
  - Zeichenketten
  - Fenster
  - Dateien
  - Programme
  - ...
- Objekte bestehen aus einem Datensatz und einer Menge von Operationen
- Nachrichten sind Aufforderungen an ein Objekt, eine Operation auszuführen
- Eine Nachricht spezifiziert nicht wie eine Operation ausgeführt werden soll, sondern welche
- Die Menge der an ein Objekt sendbaren Nachrichten bilden sein Interface
- Die internen Daten eines Objektes können nur über seine Operationen verändert werden
- Nachrichten sind der einzige Weg, die Operationen eines Objektes aufzurufen
- Die Internas eines Objektes hängen nicht von denen anderer ab; damit wird Modularität unterstützt

3.3.3 Klassen, Instanzen und Methoden

- Klassen beschreiben die Implementierung von Objekten
- Objekte mit gleichem Datensatz und Operationen gehören der gleichen Klasse an
- Objekte einer Klasse nennt man auch ihre Instanzen
- Eine Klasse beschreibt den Aufbau des Datensatzes und der Operationen ihrer Instanzen
- Der Datensatz stellt sich als Menge von Instanz-Variablen dar
- Die Werte der Instanz-Variablen sind Verweise auf Objekte

- Betrachtet man die Operationen im Hinblick ihres Aufbaus, nennt man sie auch Methoden
- Eine Methode beschreibt wie eine Operation ausgeführt werden soll, wenn sie durch eine entsprechende Nachricht aufgerufen wird
- Eine Methode beschreibt mögliche Veränderungen am Datensatz ihres Objektes
- Eine Methode spezifiziert auch ein Objekt, das als Wert der aufrufenden Nachricht zurückgegeben wird

3.3.4 Zur Syntax von Objekten und Nachrichten

Es gibt vier Typen von Ausdrücken:

1. Literale
2. Variablen-Namen
3. Nachrichten-Ausdrücke
4. Block-Ausdrücke

Literale

Literale können – sein:

1. Zahlen – in dezimaler, oktaler, hexadezimaler und weiteren Notationen
2. Zeichen – der Form `$a, $_`
3. Zeichenketten – der Form `'the Smalltalk-80 system'`
4. Symbole – der Form `#M63`
5. Felder – der Form `#(1 2 'hallo')`

Variablen-Namen

Variablen dienen der Speicherung von Objekten. Die Variablennamen kann man als Zeiger auf die Objekte betrachten. Ein Variablenname ist eine Folge von Buchstaben und Zahlen beginnend mit einem Buchstaben. Es gibt zwei Typen:

Private Variables Erreichbar nur innerhalb eines Objektes; beginnen mit kleinen Buchstaben

Shared Variables Erreichbar für mehr als ein Objekt; beginnen mit großen Buchstaben

Als Pseudo-Variablennamen gelten:

- nil
- true
- false

Nachrichten-Ausdrücke

Nachrichten-Ausdrücke bestehen aus drei Teilen: Empfänger Objekt, Selektor, Argumententeil. Es gibt drei verschiedene Nachrichten-Ausdrücke:

Unary Messages Nachrichten ohne Argumente, z. B. : `20 factorial`

Keyword Messages Nachrichten mit einem oder mehreren Argumenten; der Selektor besteht aus mehreren Schlüsselwörtern mit Doppelpunkt, die ihren jeweiligen Argumenten vorausgehen; als Beispiel:
`'The quick brown' copyFrom: 4 to: 9`

Binary Messages z. B. `3 + 4, total <= max`

Parsing Rules (Auswertungsregeln) sind:

- Unäre Ausdrücke werden von links nach rechts geparst
- Binäre Ausdrücke werden von links nach rechts geparst
- Binäre Ausdrücke haben Vorrang vor Keyword Ausdrücken
- Unäre Ausdrücke haben Vorrang vor binären
- Geklammerte Ausdrücke haben Vorrang vor unären

Cascading: Es ist möglich mehrere Nachrichten durch Semikolon getrennt an ein Objekt zu senden:

```
Empfaengerobjekt Selektor1 Argumente1;
                 Selektor2 Argumente2;
                 ...
                 SelektorN ArgumenteN.
```

Block-Ausdrücke

- Bestehen aus einer Folge von Ausdrücken, abgetrennt durch Punkte
- Nötig, um Kontrollstrukturen zu implementieren
- Blöcke sind Objekte

- Ist eine Berechnungsvorschrift, die einer Variablen zuweisbar ist
- Wird erst durch die Nachricht `value` ausgeführt; Prinzip der verzögerten Ausführung; flexibel aber ähnlich gefährlich wie goto-Befehle
- Der Form: `[Ausdruck1. Ausdruck2. ... AusdruckN]`
- Alternative Form: `[:Variable | Ausdruck1 ... AusdruckN]` (Blockargumente)

Kontrollstrukturen sind:

- Vergleichsnachrichten (binäre Nachrichten): `=, >=, >, <, <=, ~=`
- Boolean-Nachrichten: `and:, or:`
- Bedingte Anweisungen:
  - `Boolescher Ausdruck ifTrue: Block`
  - `Boolescher Ausdruck ifFalse: Block`
  - `Boolescher Ausdruck ifTrue: Block1 ifFalse: Block2`
- Schleifen
  - `Integerwert timesRepeat: Block`
  - `Block whileTrue: Block`
  - `Block whileFalse: Block`
- Iteratoren
  - to-do Iterator: `Intwert1 to: Intwert2 do: Block`
  - to-by-do Iterator: `Intwert1 to: Intwert2 by: Incr do: Block`
  - do: Iterator: `CollObjekt do: Block` Iteriert über ein Objekt der Klasse „Collection“ und gibt jedes Element an den Block weiter
  - select: Iterator: `CollObjekt select: Block` Iteriert über CollObjekt und gibt alle Elemente zurück, für die der Block ein true liefert
  - reject: Iterator: `CollObjekt reject: Block` Wie select, jedoch werden die Elemente zurückgegeben, für die der Block false liefert
  - collect: Iterator: `CollObjekt collect: Block` Iteriert über CollObjekt und gibt alle Rückgabewerte des Blocks zurück

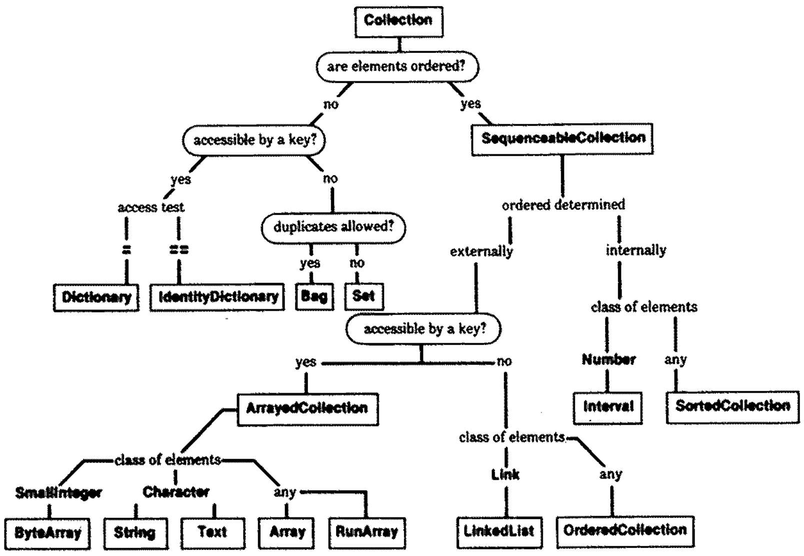


Abbildung 3.3: Der Collection-Teilbaum in Smalltalk

3.3.5 Klassen- und Methodenimplementation

Klassendefinition

Eine Klasse wird foldendermaßen definiert:

```
Superclassname subclass: #Classname
        instanceVariableNames: ' ... '
        classVariableNames: ' ... '
        poolDictionaries: ' ... '
```

Dabei sind „Superclassname“ und „Classname“ die Namen von Oberklasse und zu definierender Klasse.

Variablen

Variablentypen sind folgende:

Instance Variables gelten nur innerhalb eines Objektes und solange es existiert

Temporary Variables Definiert durch `| Variablenname |` oder es sind Blockvariablen; existieren nur für eine bestimmte Aufgabe

Class Variables können von allen Instanzen einer Klasse benutzt werden

Global Variables können von allen Instanzen aller Klassen benutzt werden

Pool Variables können von allen Instanzen einer Teilmenge von Klassen benutzt werden

Methodendefinition

Methoden werden folgendermaßen definiert:

```
MKomponente1: Argument1 MKomponente2: Argument2 ...
        "...Kommentar..."
        Anweisungen
```

Es gibt drei Arten von Methoden:

Instance Method implementieren die Nachricht auf eine Instanz der Klasse

Class Method implementieren die Nachricht auf die Klasse, bzw. auf das Klassenobjekt

Primitive Method sind atomare Methoden, die sich also nicht aus anderen zusammensetzen; es sind Operationen, die direkt von der virtuellen Maschine ausgeführt werden; ihre Form ist: `<primitive #>`; das Zeichen # repräsentiert eine Zahl

Mit der speziellen Variablen „self“ wird sich innerhalb von Methoden auf das Empfängerobjekt der entsprechenden Nachricht bezogen.

3.3.6 Vererbung

- Jede Klasse hat eine Oberklasse
- Jede Klasse hat möglicherweise eine oder mehrere Unterklassen
- Klassen weiter oben in der Hierachie repräsentieren allgemeinere Eigenschaften
- Klassen weiter unten in der Hierachie speziellere
- Vererbung aller instance variables der Oberklasse bedeutet, daß diese Variablen auch für Instanzen der abgeleiteten Klassen verfügbar sind
- Vererbung von Methoden: Zuerst wird in der Klasse selbst nach der Methode gesucht. Ist sie nicht dort, wird in der Oberklasse gesucht. Ist die Klasse „Object“ erreicht, die Methode aber nicht vorhanden, kommt es zur Fehlermeldung
- Die mehrfache Vererbung in Smalltalk ist nicht eindeutig möglich
- Überschreiben von Methoden bedeutet, daß eine bestehende Methode in einer gegebenen Klasse in einer Unterklasse neu spezifiziert wird; zur gleichen Nachricht existieren dann zwei verschiedene Methoden

3.3.7 Eine Binärrelation als Beispiel

So wird die Klasse „BinaryRelation“ definiert; mit der Oberklasse „Set“:

```
Set variableSubclass: #BinaryRelation
        instanceVariableNames: ''
        classVariableNames: ''
        poolDictionaries: ''
```

Der Klasse „BinaryRelation“ gehören zwei Methoden an:

```
addpair: aPair
        "Fuegt ein Paar als Element der Binaerrelation hinzu.""

     2 = aPair size
                ifTrue:[self add: aPair]
                ifFalse:[^false].

basicSet
        "Ermittelt die der Relation zugrundeliegende Menge."

        | concate |
        concate := #().
        self do: [ :elem | concate := concate , elem].
        ^concate asSet.
```

Dies ist die Klassendefinition einer Halbordnung:

```
BinaryRelation variableSubclass: #QuasiOrder
        instanceVariableNames: ''
        classVariableNames: ''
        poolDictionaries: ''
```

Eine Methode der Klasse „QuasiOrder“ ist:

```
addpair: aPair
        "Fuegt ein Paar als Element der Halbordnung zu, sowie die
         zugehoerigen reflexiven Paare."

   | aArray1 aArray2 |
   2 = aPair size
      ifTrue:[self add: aPair.
             (aPair at: 1) ~= (aPair at: 2)
                ifTrue:[
                   aArray1 := Array with: (aPair at: 1)
                                    with: (aPair at: 1).
                   aArray2 := Array with: (aPair at: 2)
```

```
                                  with: (aPair at: 2).
                      self add: aArray1;
                           add: aArray2.
                   ]
                ]
        ifFalse:[^false].
```

Eine Methode zur Bildung des transitiven Abschlusses fehlt noch, um die Klasse „QuasiOrder“ zu einer wirklichen Halbordnung werden zu lassen.

3.3.8 Internas und Implementation

Ein Smalltalk-80-System besteht aus zwei Teilen:

Virtual Image Es enthält alle Objekte des Systems:

- 220 Klassen
- 4500 Methoden
- 32000 Instanzen

Virtual Maschine Ihre Eigenschaften:

- Sie verarbeitet eine eingeschränkte Untermenge von Smalltalk
- Verwaltet werden hier auch die Gerätesteuerung und der Microcode

Compiler

- Methoden im Quellcode sind Instanzen der Klasse „String“
- Der Compiler übersetzt die Methoden in Bytecode
- Der Bytecode ist eine Instanz der Klasse „CompiledMethod“
- Er (der Bytecode) wird im „Dictionary“ der Klasse gespeichert, der die Methode angehört.

Interpreter

- stackorientiert
- versteht 256 Bytecodes, eingeteilt in fünf Kategorien:

  push legt Objekt auf den Stack

  store legt den aktuellen Wert des Stacks in eine Variable

  send legt einen Message-Selektor und die Anzahl der Argumente fest

  return zur Beendigung einer Methode

  jump Sprunganweisung zu einem anderen Bytecode

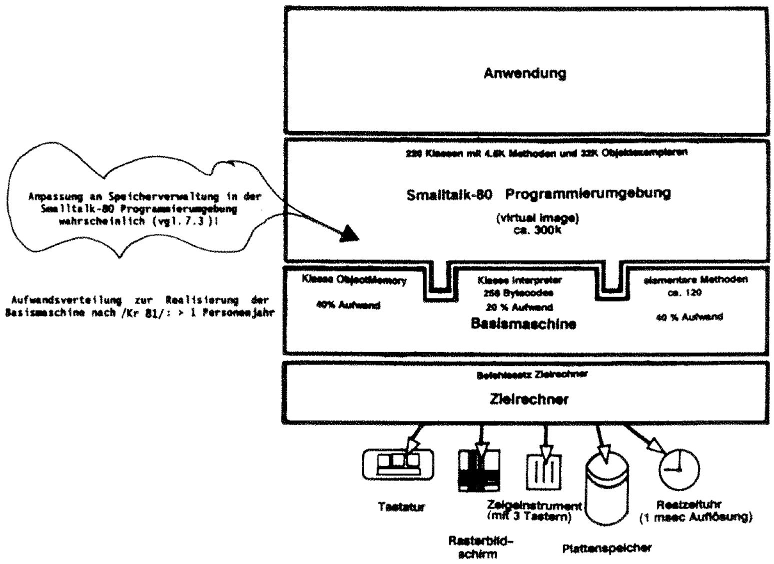


Abbildung 3.4: Smalltalk-80 Systemarchitektur

Erfahrungen, Probleme, Ausblick

Erfahrungen

- Schnelles Einarbeiten und leichtes Produzieren von Programmen ist schwierig
- Da das System völlig transparent ist, muß man sich im Ganzen mit ihm beschäftigen; das braucht Zeit
- Smalltalk-80 empfindet man als einen Softwarebaukasten; programmiertechnische Standardkonzepte (Liste, Beutel, ...) sind vorgefertigt und man muß sich mit ihnen schon zu Anfang beschäftigen

Probleme

- Ungelöst ist die Frage, wie man mit der fehlenden Typisierung umgehen soll, bzw. kann man sie einführen
- Eine statische Bindung schon während der Übersetzung von Botschaftsausdrücken ist immer noch nicht möglich
- Das Konzept der Mehrfachvererbung wird nicht eindeutig gefördert

Ausblick

- Smalltalk wird zum Softwareprototyping eingesetzt; nach dem Testlauf wird das Programm in eine schnellere Sprache übersetzt
- Anwendung scheint es nicht so sehr im wissenschafts-, als vielmehr im verwaltungstechnischen Bereich zu finden
- Es gibt Standardisierungsbestrebungen nach ANSI X3J20
- Wichtiges Derivat ist „VisualWorks“

3.4 C++

Anja Fröhner, Nils Schmeißer

3.4.1 Allgemeine Einführung

Zur Entwicklung von C++

Frühe Versionen der Sprache, meist als " C mit Klassen " bezeichnet, existieren bereits seit 1980. Die Sprache wurde ursprünglich von Bjarne Stroustrup erfunden, um ereignisgesteuerte Simulationen schreiben zu können, für die SIMULA67 ideal gewesen wäre, abgesehen von der Effizienz. C++ wurde dann an verschiedenen Stellen entwickelt, 1989 einigte man sich auf einen ANSI-Standard für C++. Seine wesentlichen Bestandteile verdankt C++ in naheliegender Weise C, aber auch SIMULA67 (das Klassenkonzept), BCPL und ALGOL68 (das Überladen von Operatoren und die freie Plazierung von Deklarationen im Programmtext). Inspiration für die Templates gaben die Ada-Generica und parametrisierte Module in Clu.
Stroustrup gibt als Entwicklungsziele an, das Schreiben guter Programme zu erleichtern und das Programmieren für den Programmierer kurzweilig zu machen, wie immer das zu verstehen sei.

C und C++

Bei der Entwicklung von C++ wurde großer Wert darauf gelegt, trotz bekannter Mängel von C die Kompatibilität aufrechtzuerhalten, denn es existieren bereits große Mengen an Quellcode in C, die einerseits von C++ profitieren könnten. Andererseits kann C++ die zahlreiche Bibliotheks- und Utility-Software von C üvernehmen. Viele Programmierer kennen C bereits, sie haben nun die Möglichkeit, C und C++ parallel zu verwenden und schleichend zur objektorientierten Programmierung zu wechseln. Einsichtig ist, daß bei einer Neuentwicklung die Unterschiede zu C entweder gravierend oder klein sein müssen. Man entschied sich für letzteres.
Was aber sind die Vorteile von C, um gerade diese Sprache als Grundlage zu wählen ? C ist vielseitig, kurz (im Sinne geringen vorgegebenen Funktionsumfanges) und relativ low-level, für Aufgaben der (UNIX-)Systemprogrammierung geeignet und auf allen Maschinen / unter allen Systemen verfügbar.
Neu aufgenommene Funktionen in C++ sind `new, delete, throw` und `try`.

Charakteristika von C++

Die wichtigsten Entwurfskriterien für C++ waren **Einfachheit**, also eine freie Sprache mit wenigen Regeln, und **C-Kompatibilität**.
Es gibt keine High-Level-Datentypen mit High-Level-Funktionen. (Die Datentypen sind dieselben wie in JAVA bis auf den Typ `boolean`.)

Im Vergleich zu C unterstützt C++ viel stärker Typisierung und Struktur. Ein typischer Aspekt dabei ist die Datenkapselung. Typprüfung erfolgt während der Übersetzungszeit, eine echte Zugriffskontrolle bzw. Typprüfung während der Laufzeit erfolgt nicht. Insbesondere die Strukturierung großer programme zu komplexen Sachverhalten wird durch die Unterstützung von Modularität, Datenabstraktion und das Klassenkonzept ermöglicht.

3.4.2 Paradigmen des Programmierens und ihre Realisierung in C++

Prozedurales Programmieren

Man benutzt **Funktionen**, um Ordnung in ein Geflecht von Algorithmen zu bringen.

1. Deklaration:

   ```
   outputtype1 function1(inputtype1 arg)
   {
   ...//Quelltext
   return result;
   }
   ```

2. Verwendung:

   ```
   void function2();
   {
   inputtype1 y = ...;
   outputtype1 x = funktion1(y);
   }
   ```

Modulares Programmieren

Module erlangen Bedeutung, wenn neben den funktionalen Algorithmen die Organisation der zugehörigen Daten eine bedeutende Rolle spielt. Man kapselt aufeinander bezogene Prozeduren mit den betreffenden Daten in Moduln. Die Daten lassen sich nur über eine vorgegebene Schnittstelle modifizieren. Hier kommt erstmalig das Geheimnisprinzip zur Anwendung: bei Benutzung der Schnittstelle weiß man nicht, was innerhalb des Moduls abläuft. Folgendes Beispiel eines Stack-Moduls soll das veranschaulichen:

1. Schnittstelle:

```
extern void push(char) ;
extern char pop();
const int stacksize = 100;
```
[2].

2. Implementation:

```
♯ include ''stack.h''
static char v[stacksize];
static char* p = v;
void push(char c);
{
//Überlauf abfangen und push
}
char pop();
{
//Unterlauf abfangen und pop
}
```

3. Verwendung:

```
♯ include ''stack.h''
void somefunction();
{
push('a');
char c = pop();
if(c! =' a') error('' impossible'');
}
```

Modula2 unterstützt diese Technik aktiv, C ermöglicht sie zumindest.

Datenabstraktion

Alle Daten eines Grundtypes werden unter der Kontrolle eines Datentypmoduls zentralisiert. Das Modulkonzept ermöglicht diese Datenabstraktion, unterstützt sie aber noch nicht genügend. Die selbstdefinierten Typen unterscheiden sich wesentlich von den vordefinierten Datentypen (weniger Möglichkeiten).
Ada, Clu und C++ erlauben abstrakten Datentypen, sich (fast) wie eingebaute zu verhalten und unterstützen damit Datenabstraktion. Ein Beispiel zur Erklärung:

[2] Diese Anweisung erzeugt eine lokale Kopie von `stacksize` in **jedem** Modul, das "stack.h" einbindet

1. Deklaration:

```
class complex {
double re, im;
public:
complex(double r, double i) {re = r; im = i}
complex(double r) {re = r; im = 0}
friend complex operator+(complex, complex);
friend complex operator-(complex, complex);
friend complex operator-(complex);
friend complex operator*(complex, complex);
friend complex operator/(complex, complex);
// ...
}
```

2. Implementation:

```
complex operator+(complex a1, complex a2)
{
return complex(a1.re + a2.re; a1.im + a2.im);
}
```

3. Aufruf:

```
void f()
{
complex a = complex(2.3);
complex b = 1/a;
complex c = a+b*complex(1.2, 3);
// ...
}
```

Das Problem : Der abstrakte Datentyp ist eine Art "black box". Die Anpassung an neue Anwendungskontexte ist nur über eine Änderung der Definition des abstrakten Datentypes möglich. Programmerweiterungen sind nur dann machbar, wenn der komplette Quelltext zur Verfügung steht. Daraus resultieren gegebenenfalls Fehleranfälligkeit und Änderungsaufwand.

Objektorientierung

Der Umfang der Gemeinsamkeiten zwischen mehreren Typen einer Anwendung ist das Kriterium für die Anwendbarkeit objektorientierter Programmierung. Wo es keine Gemeinsamkeiten gibt, genügt Datenabstraktion. Ein Beispiel in C++:

1. Basisklasse:

```
class shape {
point center;
color col;
public:
point where() {return center};³
void move(point to) {center = to; draw();}
virtual void draw();
virtual void rotate();
};
```

   Auf `virtuell` gekennzeichnete Methoden kann das Verfahren der späten Bindung angewendet werden.

2. Abgeleitete Klasse:

```
class circle : public shape
{
int radius;
public:
void draw() { //...};
void rotate() { };
};
```

Dabei macht `puplic` eine Aussage über die Zugreifbarkeit der deklarierten Funktionen von außen.

3.4.3 Unterstützung für Datenabstraktion

Initialisierung, Zuweisung, Zerstörung

Das Klassenkonzept findet sowohl bei der Datenabstraktion als auch bei der objektorientierten Programmierung in C++ Anwendung. Der Unterschied besteht darin, daß im ersten Fall keine abgeleiteten Klassen existieren. Objekte einer Klasse (eines Datentyps) werden mittels des `constructors` initialisiert und allokiert (Definition/Deklaration). In C++ trägt der Konstruktor denselben Namen wie die Klasse selbst. Bei allen nichttrivialen Objektkonstruktionen ist eine Cleanup-Funktion (`destructor` sinnvoll, um allokierten Speicher freizugeben etc. Er erhält den Klassennamen mit Präfix $\sim$. Als Beispiel:

[3] die Definition von Methoden in der Objekt-Definition impliziert eine sogenannte Inline-Expansion, d.h. der Code der Methode wird nicht über einen Unterprogrammaufruf aktiviert, sondern die Maschinenanweisungen werden an der Stelle ihrer Nutzung im Modul expandiert (eingebunden). Das bringt einen Geschwindigkeitsvorteil. Es wird aber davon abgeraten (zumindest in der Testphase), um Fehler vermeiden zu können.

1. Deklaration:

```
class vector {
int sz;
int* v;
public:
vector(int);          //Konstruktor
~vector();          //Destruktor
int& operator[](int index);
};
```

2. Implementation:

```
vector::vector(int s)
{
if(s<=0) error(''bad vector size'');
sz = s;
v = new int[s];          //allokiert ein Array mit s
int-Werten
}
vector::~vector()
{ delete[] v;          //deallociert das Array v
};
```

In C++ werden Speicherbereiche, auf die kein Zeiger mehr zeigt, nicht automatisch freigegeben (keine automatische Garbage-Collection)[4]. Stattdessen erhält ein Typ über den Konstruktor/Destruktor-Mechanismus die Möglichkeit, seine eigene Speicherverwaltung zu übernehmen.
Ein Konstruktor der Form `T(const T&)` definiert alle Initialisierungen eines Objektes vom Typ T mit einem anderen Objekt vom Typ T. Es ist der sogenannte Copy-Konstruktor. Beispiel der Implementation:

```
vector::vector(const vector& a)
{
sz = a.sz;
v = new int[sz];
for (int i=0; i<sz; i++) v[i] = a.v[i];
} ;
```

Ein geeignetes Instrument zur Zuweisung von Datentypen ist auch die Überlagerung von =.
Wir verweisen an dieser Stelle auf die drei unterschiedlichen Mechanismen zur Übergabe von Parametern an Funktionen:

[4] Die C++ Spezifikation verbietet das nicht. Laut [Str93] gibt es Implementationen von C++, die über eine Garbage-Collection verfügen.

1. **Wertparameter**: `(typ parameter)`, hier wird auf dem Stack eine Kopie der übergebenen Instanz erzeugt, mit der im Weiteren gearbeitet wird

2. **Variablenparameter**: `(typ *parameter)`, hier wird ein Zeiger auf die Parameterinstanz als Wertparameter übergeben

3. **Referenzparameter** (das ist neu in C++): `(typ &parameter)`, es wird eine Referenz auf das Argument übergeben, in der Funktion wird also auf das Argument *an sich* zugegriffen

Templates

Benötigt man mehrere Varianten eines Types (z.B. Vektoren mit verschiedenen Elementtypen), bietet es sich an, ein Template zu definieren. Ein Template legt fest wie eine Familie verwandter Klassen erzeugt werden kann. Ein Beispiel:

```
template<class T> class Vector {
T* v;
int sz;
public:
Vector(int s)
{
if(s<=0) error(''bad vector size'');
v = new T[sz = s]; }
T& operator[](int i);
int size() {return sz;}
// ...
};
```

Nun können Vektoren eines speziellen Types wie folgt vereinbart und benutzt werden:

```
void f()
{
Vector<int> v1(100);
Vector<complex> v2(200);
v2[i] = complex(v1[x], v1[y]);
// ...
}
```

Vergleichbares zu Templates wird in anderen Sprachen "parametrisierte Typen" oder "Generics" genannt, etwa in Clu oder Ada gibt es solche. Die Benutzung von Templates führt nicht zu Laufzeitverlusten (Stichwort: Präprozessor).

Abstrakte Klassen, Mehrfach-Implementationen

Man kann eine Basisklasse deklarieren, deren Funktionen (`virtual`) zunächst leer sind. Sie gibt dann den Rahmen für einen Typ vor, der im einzelnen unterschiedlich implementiert wird. Die verwendeten Funktionen werden deklariert nach dem Schema :

```
virtual void push(T) = 0;          //leere virtuelle Funktion
```

Eine abstrakte Klasse ist ausschließlich als Basisklasse verwendbar.

3.4.4 Unterstützung des objektorientierten Programmierens

Die objektorientierten Sprachelemente erlauben im Vergleich zu Sprachelementen, die ausschließlich für Datenabstraktion entworfen wurden, weitaus flexiblere und allgemeinere benutzerdefinierte Typen.

Aufrufmechanismen

Es gibt generell zwei Möglichkeiten, eine Elementfunktion eines Objektes aufzurufen. Ist das Objekt **obj** vom Typ **T**, dann wird auf die Elementfunktionen mit

```
obj.funktion
```

zugegriffen. Wird **p** als Zeiger auf ein Objekt einer Klasse **T** deklariert (`T* p`), dann ruft man die Elementfunktionen des Objektes mit:

```
p→funktion
```

auf. Die statische Typprüfung zur Übersetzungszeit stellt sicher, daß das Programm in seiner Typverwendung so konsistent ist, wie vor der Ausführung festgestellt werden kann.
Besondere Aufmerksamkeit erfordert der Aufruf virtueller Funktionen. Der Aufrufmechanismus muß im Objekt Informationen des Compilers herausfinden, welche Ausführung der Funktion aufgerufen wird. Die Semantik des Aufrufes bleibt dieselbe.
Die statische Typprüfung in C++ bedeutet gewisse Einschränkungen. Durch Kombination von Templates und Vererbung kann durch Kombination von Templates und Vererbung eine vergleichbare Flexibilität und Handhabbarkeit erreicht werden wie in Sprachen mit Laufzeittypprüfung. Allerdings zwingt dies zu einen anderen Programmierstil als dynamische Typprüfung. In C++ stellen Klassen (im Gegensatz zu Smalltalk) eine verbindliche Spezifikation dar, wobei dem Anwender garantiert wird, daß der Compiler nur die in der Klassendefinition angegebenen Operationen akzeptiert.

Mehrfache Vererbung

Mehrfache Vererbung bedeutet: Die Klasse **B** erbt Eigenschaften sowohl von der Basisklasse **A1** als auch von der Basisklasse **A2**.
C++ unterstützt mehrfache Vererbung, ohne daß sie mit signifikanten Laufzeit- oder Speicherplatzkosten verbunden ist. Schematisch stehen folgende Deklarationen zur Verfügung:

```
class B : public A1, public A2 {
//zusätzliche Elemente
};
```

Mehrdeutigkeiten werden zur Übersetzungszeit behandelt, d.h. wenn in **A1** und **A2** jeweils eine Funktion gleichen Namens, aber verschiedenen Inhaltes existiert, dann wird diese als in **B** nicht definiert betrachtet. Bei qualifiziertem Zugriff in der abgeleiteten Klasse, etwa `A1::Methode` oder `A2::Methode` gilt der Namenskonflikt als aufgelöst. Von besonderem Interesse ist hier das Konzept der *virtuellen Basisklasse*

```
class A1 : virtual public A {
...
};

class A2 : virtual public A {
...
};

class B : public A1,public A2 {
...
};
```

Wird `A` als virtuelle Basisklasse erklärt, so werden die Attribute und Methoden von `A` für `B` nur einmal erzeugt. Im Gegensatz dazu existieren bei nicht-virtueller Basisklasse `A` jeweils Komponenten `X` mit `B::A1::A.X` und `B::A2::A.X`, die **voneinander verschieden** sind[5].

Kapselung

Datenelemente oder Elementfunktionen einer Klasse sollen manchmal vor unautorisiertem Zugriff geschützt werden. Dann können alle abgeleiteten Klassen auf die geschützten Elemente zugreifen, nicht erfolgen kann der Zugriff in umgekehrter Richtung oder aus anderen Zweigen des Vererbungsbaumes. Ein Beispiel:

[5]Die Existenz zweier Instanzkomponenten kann manchmal gewollt sein.

```
class window {
// ...
protected:
Rectangle inside;
// ...
};

class dumbterminal : public window {
// ...
public:
void prompt();
// ...
};
```

Hier kann eine Elementfunktion der abgeleiteten Klasse - etwa `dumbterminal::prompt()` - auf `window::inside` zugreifen, andere Funktionen nicht.
Je größer der Programmumfang, desto mehr gewinnt Datenkapselung an Bedeutung. Für den Bedarfsfall sei hier das Stichwort **private** erwähnt. Zugriffsrechte werden gemäß der folgenden Tabelle vererbt:

Vererbung	private	protected	public
private	private	private	private
protected	private	protected	protected
public	private	protected	public

3.4.5 Praktische Hinweise

Programmgestalt

Ein komplexes C++-Programm gliedert man im allgemeinen in eine größere Anzahl von Teilprogrammen. Unbedingt dabeisein muß eine Funktion `main()`, deren Ausführung das Programm startet. Im Programmkopf werden alle (selbst)deklarierten Schnittstellen eingefügt:

```
♯include <iostream.h>
♯include ''stack.h''
//...
```

Es ist üblich, Teilprogramme (Module) für verschiedene Sachverhalte zu schreiben, in Dateien der Form `teil1.h` die Deklaration der Schnittstelle zu fassen und in `teil1.cc` die Implementation durchzuführen. Die Module werden nach dem Kompilieren gebunden (gelinkt). Ein nützliches Werkzeug dazu stellt `make` dar, welches z.B. in der UNIX-Welt oft angewandt wird (nähere Ausführungen zu `make` finden sich in A.4).

3.5 Java

Andreas Schälicke

3.5.1 Ursprung und Eigenschaften

Ursprung

Die Entwicklung von Java began 1990 bei der Firma Sun unter der Leitung von James Gosling. Zu diesem Zeitpunkt trug es aber noch den Namen "Oak"[6], und sollte eine neue Hochsprache darstellen, die insbesondere zur Programmierung "hybrider Systeme im Consumer-Bereich"[7] geeignet sein sollte. In diesem Bereich hat die Sprache "Oak" aber ke inen Erfolg.
Anfang 1993 waren weltweit ca. 50 WWW-Server bekannt und der "WWW-Boom" begann gerade. Sun erkannte dann, daß "Oak", inzwischen umbenannt in "Java", durch seine Portabilität ideal für Internet-Anwendungen geeignet war. So wurde ein Konzept ausgea rbeitet, um Java-Programme in HTML-Seiten zu integrieren. Hierzu wurde von SUN "Hot-Java" entwickelt, der als erster Browser Java-Applets untersützte.
Heute werden Java-Script und Java-Applets von allen modernen Browsern unterstützt, und Entwicklungssysteme für Java-Programme sind für alle gängigen Plattformen verfügbar.

Eigenschaften von Java

Java behauptet von sich

- portabel
- objektorientiert
- multithreading fähig
- sicher / robust

und

- einfach

zu sein.

[6]Zu diesem Namen soll es gekommen sein, als J. Gosling vor seinem Fenster eine Eiche (engl. oak) sah .
[7]z.B. Waschmaschinen, Telefonanlagen oder Toster

Unterschiede zu C++

Ursprünglich wollte J. Gosling C++ nur um einige Fähigkeiten erweitern, erkannte aber, daß Einfachheit und Sicherheit so in C++ nicht realisierbar sein würden. Trotzdem lehnt sich Java in seinem äußeren Erscheinungsbild sehr an C++ an. Die wichtigst en Unterschiede sind:

- keine Typedefs, Defines und kein Präprozessor
- keine Strukturen oder Unions (also nur Klassen)
- keine Funktionen außerhalb von Klassen (nicht mal main())
- keine Multiple Vererbung
- keine Goto-Befehle
- keine Automatische Typenumwandlungen
- keine Pointermanipulationen oder manuelle Speicherverwaltung

Die virtuelle Maschine

Der Quelltext (Unicode-Escape-Sequenzen möglich) [8] wird erst in den Unicode-Text übertragen. Dann wird dieser Text compiliert und der Byte-Code erzeugt. Der Byte-Code kann dann zur Laufzeit, zumindest wenn die benötigten Bibliotheken verfügbar sind, durch einen Browser (oder einen anderen Java-Byte-Code-Interpreter) in den dann plattformabhängigen Code übersetzt werden. Die Übersetzung durch Compiler ist ebenfalls möglich und auch Java-Prozessoren sind (waren) in Entwicklung.

3.5.2 Einfache Datentypen und Konstrukte

Einfache Datentypen

Obwohl Java fast vollständig mit Objekten/Klassen arbeitet, werden einige grundlegende einfache Datentypen bereitgestellt (alle Variablen werden automatisch initialisiert):

Typ	*Wertebereich*	*Initialwert*
`char`	Unicode-Zeichen	`'\u0000'`
`byte`	-128..127	`(byte)0`
`short`	signed 16 Bit	`(short)0`
`int`	signed 32 Bit	`0`
`long`	signed 64 Bit	`0L`
`float`	32 Bit IEEE	`0.0f`
`double`	64 Bit IEEE	`0.0`
`boolean`	false/true	`false`

[8] z.B. ist `\u00DC` gleichbedeutend mit `Ü`, beide Varianten können im Quelltext auftauchen

Es existieren aber zu jedem dieser Datentypen Klassen, die die Funktionalität bereit stellen. Für `int` ist es `Integer`, für `char` ist es `Character`, für `float` `Float` usw.
Für den Umgang mit Strings sind die Klassen `String` und `StringBuffer` vorhanden. Es kann aber eine vereinfachte Schreibweise verwendet werden:

```
String s ="Text A";
```

ist gleichbedeutend mit

```
String s =  new String("Text-A");
```

Einfache Anweisungen

Die if-Abfrage

```
if (a <b)
 System.out.println("kleiner");
else
 System.out.println("kleiner");
```

Die while-Schleife

```
// Applet Cursorform aendern
Object frame= getParent();
while(!(frame instanceof Frame))
 frame =
  ((Component)frame).getParent();
((Frame)frame).setCursor(
   Frame.HAND_CURSOR);
```

Die do-Schleife

```
do {
 ...
 i++;
} while (i<10);
```

Die for-Schleife

```
for (int i=1,j=10; i<9; i++, j--)
 System.out.println(i+" "+j);
// weil Info optional moeglich:
for ( ; ; ) ...;
// entspricht:
while( true ) ...;
```

Die switch-Verzweigung

```
switch(x) {
 case 3:
  System.out.println("Drei");
 case 4:
  System.out.println("Drei o. Vier");
  break;
 case 6:
  System.out.println("Sechs");
  break;
 default:
  System.out.println("was anderes");
}
```

Schleifen mit Sprung-Marken

```
demo: { index = 0;
 while(index<max) {
  if (a[index++]==0)
   break demo;
  } //hierher ohne demo
} //mit marke hierher

demo2: while(flag) {
 // Fortsetzung hier
 index = 0;
 while( index < 10)
  try{
  if( !check(index++))
   countinue demo2;
  } finally {
    // hier auch bei continue
  }
}
```

3.5.3 Objektorientierte Programmierung mit Java

Man kann keine Java-Programme schreiben ohne objektorientiert zu programmieren. Um die Programme besser lesbar zu gestalten, gelten in Java folgende (nicht zwingend bindene) Konventionen:

- “sprechende Bezeichner” verwenden (außer `i`, `j` in Schleifen)
- Namen einfacher Datentypen werden vollständig klein geschrieben (geht gar nicht anders)
- Konstanten (`final` Elemente) werden vollständig groß geschrieben (z.B.: `ChessMan.WHITE`)
- Klassen werden mit großen Anfangsbuchstaben und sonst bis auf Anfangsbuchstaben der Teilwörter klein geschrieben (z.B.: `ChessFrame`)
- Variablen und Methoden werden mit kleinen Anfangsbuchstaben und sonst bis auf die Anfangsbuchstaben von Teilwörtern klein geschrieben (z.B.: `ChessFrame.whiteIsOnMove`)

Klassendefinition

Klassen werden mit dem Schlüsselwort `class` gefolgt vom Namen der Klasse eingeleitet:

```
public class ChessFrame extends Frame {
  Applet startApplet;              // Instanz auf Applet.class
  Image background;                // Instanz auf Image.class

  ...
  boolean whiteIsOnMove=true;      // Weiss begint
  int moveCount=1;                 // mit dem 1. Zug
  boolean gotStart=false;          // Noch kein StartPunkt fest
                                   // gelegt
  public ChessFrame(String title, Applet startApplet) {
  // kein void weil Konstructor
    super(title);
  // Eltern-Konstruktoren koennen nur als erster Befehl
  // ausgefuert werden
    this.startApplet= startApplet;
  // ueberdecken lokale Variablen die globalen
  // kann mit this auf diese zugegriffen werden

  ... // weitere Initialisierungen
    startPoint= new Point(0,0);
```

```
  }

  public boolean mouseDown(Event evt, int x,int y) {
    ... // Implementation
    return true; // Ergebnis zurueckliefern
  }

  ... // weitere Methoden oder Variablen
}
```

Konstruktoren haben keinen Ergebnistyp, auch nicht `void`, und müssen den selben Namen wie die Klasse tragen. Es können mehrere Konstruktoren implementiert werden, wenn jederzeit eine eindeutige Zuordnung anhand der Parameter möglich ist (Funkti onsüberladung).

Vererbung

Java unterstüzt eine einfache Vererbung bei Klassen, aber eine mehrfache Vererbung von Schnittstellen. Die Elternklasse wird hinter dem reservierten Wort `extends` angegeben, Schnittstellen hinter `implements`

```
public class Chess extends Applet implements Runnable, GraphObj {
  ...
```

Die Deklaration von Schnittstellen verläuft ähnlich der der Klassen, und diese sind praktisch abstrakte Klassen mit nur abstrakten Methoden. Eine Instance vom Typ eines Interface kann auf alle Klassen verweisen, die dieses Interface implementieren.

```
public interface GraphObj {
  public void setGreen();
  ...
}
```

Modifier

Durch den Vorsatz eines Modifiers können die Eigenschaften (z.B.: Zugriffsrechte) von Variablen, Methoden und Klassen verändert werden:

final Mit `final` ist es möglich die Veränderung eines Elements zu verhindern. Variablen werden zu Konstanten, Methoden können nicht überschrieben werden und Klassen können nicht als Elternklassen verwendet werden.

static Mit `static` werden Elemente (vorwiegend bei Konstanten, also zusammen mit `final`), oder Methoden gekennzeichnet, auf die ohne Instanzierung der Klasse zugegriffen werden können soll.

volatile `volatile` veranlaßt den Compiler, Vorsichtsmaßnahmen zur Gewährleistung ständiger Konsistenz des Elementes zu treffen.

transistent Dieser ist bisher nicht benutzt und soll Elemente kennzeichnen, die nicht abgespeichert werden müssen, da sie keinen Zustand beschreiben.

abstract Klassen mit diesem Modifier können nicht instanziert werden. Methoden, die (noch) nicht implementiert werden, tragen diesen Bezeichener. Sobald eine Klasse eine als `abstract` deklarierte Methode besitzt, muß auch die Klasse `abstract` sein.

public `public` macht eine Methode für alle Klassen und Pakete sichtbar.

protected Abgeleitete Klassen (auch wenn nicht im selben Paket) und alle Klassen des selben Paketes können auf eine so definierte Methode zugreifen.

private protected Diese Zugriffsklasse entspricht dem `protected` in C++. Der Zugriff ist also für alle Nachfahren erlaubt, sonst nicht (nicht mehr ab Ver. 1.1).

private Methoden mit diesem Modifier können nur in der Klasse verwendet werden, in der sie definiert sind.

Programme in Java: Applets oder Applicationen

Der Aufbau eines Programmes ist abhängig davon, ob eine Application, also eine Anwendung für den direkte Aufruf auf der Zielplattform, oder ob ein Applet, also eine Anwendung die durch einen Webbrowser gestartet werden soll, entwickelt wird.

Applicationen

Eine Application entspricht einem normalen Programm, wie sie auch mit anderen Programmiersprachen entwickelt werden. Die Hauptklasse der Application muß die Methode

```
  public static void main(String args[])
```

implementieren. Innerhalb dieser statischen Methode wird in der Regel erst eine Instanz auf die eigene Klasse angelegt, damit auf die nichtstatischen Teile des Programmes zugegriffen werden kann.

```
import java.awt.Frame;

public class FrameApplication extends Frame {
  public static void main(String args[]) {
```

```
     FrameAplication fa = new FrameApplication();
     fa.setSize();    // ohne Instanzierung ist ein Zugriff auf
     fa.show();       // setSize() nicht moeglich
  }
  public void setSize() {
    resize(200,2000);
  }
}
```

Applets

Ein Applet besitzt die besondere Fähigkeit in HTML-Seiten integriert zu werden. Dieses eröffnet ungeahnte Möglichkeiten seine WWW-Seiten zu verändern. Zum Einbinden in die HTML-Seite wurde ein `APPLET`-Tag definiert:

```
<APPLET CODE=Chess.class WIDTH=200 HEIGHT=100>
</APPLET>
```

Die Klasse eines Applets muß von der Klasse `Applet` abgeleitet sein. Die Aufgabe des Initialisierungsteiles übernimmt die Methode `void init()`. Weiterhin sind die Methoden `void start()`, `void stop()` und `void dest roy()` für den Ablauf des Applets wichtig.

```
import java.applet.Applet;

public class AppletDemo extends Applet {
  public void init() {
    // Initialisierung
  }

  public void start() {
    // Aufruf beim Betreten einer HTML-Seite
  }

  public void stop() {
    // Aufruf beim Verlassen der HTML-Seite
  }

  public void destroy() {
    // Aufruf bei einer Speicher-frei-Aktion des Browsers
  }
}
```

Two in One

Diese beiden Programm-Varianten wiedersprechen sich nicht, so daß die Klasse sowohl als Applet, als auch als Application läuft. Dazu muß nur die Klasse von der Klasse `Applet` abgeleitet sein und die Methode `public static void main(Strin g[] args)` geeignet implementiert werden.

3.5.4 Eigene Erfahrungen

Java ist auf jedenfall nicht so *sicher* oder *einfach* wie immer behauptet wird. Natürlich hat die Sprache viele Ansätze, die für eine sichere Programmierung vorteilhaft sind, aber viele Implementationen (z.B. in Netscape) sind *unausge reift*. Die Fehlermeldungen vom Javacompiler sind vielleicht einleuchtender als manche Fehlermeldungen in C++, aber für mich, ursprunglich Turbo Pascalanwender, waren diese doch eher verwirrend als hilfreich.
Mit der Geschwindigkeit der Compilierung unter Linux war ich zufrieden, da das Programm ja in einzelne Klassen-Quelldateien zerlegt werden kann (oder muß), und so nur veränderter Quellcode neu übersetzt werden muß. Die Ausführungsgeschwindigkeit im lokalen Appletviewer war zügig, mit dem Laden übers Netz steigt natürlich die Initialisierungszeit beträchtlich.
Mit Java lassen sich relativ schnell graphische Elemente erzeugen und in WWW-Seiten integrieren. Die mitgelieferten Bibliotheken sind umfangreich und nützlich. Zum Lösen von mathematischen Problemen ist Java aber kaum geeignet.
Mein Testprogramm (`chessapplet.html`) findet man unter:

`http://www.phy.tu-dresden.de/~as10/java/}`

3.6 Ada 95

Nils Schmeißer

[Bar82] schreibt über Ada: "Ada ist eine höhere Programmiersprache, die ursprünglich vom US Verteidigungsministerium (USMD) gefördert wurde, um in dem sogenannten *integrierten Systemanwendungsgebiet* Verwendung zu finden[9]. Die Entwicklung von Ada wurde 1974 durch eine Studie des USMD eingeleitet. Auf Grundlage zweier Papiere (Strawman, 1975 und Tinman, 1976) wurden die vorhandenen Sprachen in drei Kategorien eingeteilt:

- "nicht passend", da veraltet oder für ein anderes Gebiet gedacht (FORTRAN, CORAL66)
- "nicht unpassend" (RTL/2, LIS)
- "empfohlene Grundlage" (Pascal, PL/1 und Algol 68)

Nach einem letzten Papier (Steelman) wurde eine Gruppe von CII Honeywell Bull Sieger und die, von dieser Gruppe entwickelte Sprache wurde 1979 unter dem Namen "Ada"[10] vorgestellt. Ada ist gemäß ANSI/MIL-STD-1815A standardisiert.
Zu den Hauptgesichtspunkten, unter denen Ada entwickelt wurde zählen

- Lesbarkeit, ist der Schlüssel für den Test, die Wartung und die Erweiterung von Software
- "starkes" Typkonzept, wir verweisen hier auch auf Pascal
- Mechanismen für Datenkapselung und Bibliotheksmanagement, das ermöglicht die Modularisierung von Code (siehe auch C++)
- Behandlung von Ausnahmebedingungen (Exceptions), auch im Fall eines fehlerhaften Unterprogrammes oder fehlerhafter Zwischenergebnisse muß das gesamte Programm nicht notwendig unterbrochen werden, es kann u.U. trotzdem noch zum richtigen Endergebnis gelangen (Beispiel: ein Steuerprogramm erhält einen fehlerhaften Meßwert, aus dem keine Reaktion abgeleitet werden kann, der nächste Meßwert hingegen ist wieder korrekt. Dann sollte das Programm an dieser Stelle nicht terminieren.)[11].
- Datenabstraktion (Trennung von Deklaration und Definition)
- Prozesse, d.h. mit Ada ist paralleles Programmieren möglich

[9] Ein integriertes System ist eines, in dem der Computer ein eingebautes Teil eines größeren Systems ist, wie z.B. einer chemischen Fabrik, einer Rakete oder eine Geschirrspülmaschine.

[10] Der Name wurde zur Ehre von Augusta Ada Byron, Gräfin von Lovelace (1816-1851) vergeben.

[11] Die Behandlung von Ausnahmebedingungen ist auch in C++ und Java vorgesehen.

- Generische Einheiten

Ein Ada Programm ist eine Folge von Einheiten, die im Rumpf einer Prozedur stehen, etwa

```
with package1,...,packagen;
use package1,...,packagen;

procedure main is
  var_dcl;
begin
  code
end main;
```

Für die Beschreibung der Syntax verweisen wir auf [Gon91], wir beziehen uns bei allen weiteren Ausführungen auf Ada95 ([CGS]).

Klassendefinition

Im Gegensatz zu Turbo-Pascal und C++, orientiert sich Ada mehr am theoretischen Modell, eine Klasse wird als Kombination aus abstraktem Datentyp und einer Menge von Methoden. Die Deklaration einer Klasse erfolgt meist getrennt von der Definition in einer Datei mit dem Namen "classname.ads"

package classname_package **is**
    – datatype
        – Definition einer Sorte
        type sort **is private**;
        – Ableitung einer öffentlich zugänglichen Sorte
        subtype subsort **is** sort;
    – methods
        procedure proc **(** args **)**;
        function func **(** args **) return** returntype;
        – Konstruktor
        function NewSubsort **(** args **) return** subsort;
    – Definition der Sorte ist privat
        private
            type sort **is**
                ...
end classname_package **;**

In der zugehörigen "classname.adb" erfolgt nur noch die Implementation der Methoden.

package body classname_package **is**
    – methods
        procedure proc **(** args **) is**
            procbody;
        end proc;
        function func **(** args **) return** returntype **is**
            funcbody;
        end func;
end classname_package **;**

Vererbung

Die Deklaration "**subtype** subsort **is** sort" ist Bildung eines Unterbereichstypen inklusive der Bildung einer impliziten Konversionsfunktion. Eine Klasse resp. ein package erbt von seinen Eltern durch die Anweisungen **with**:

with parent_package_1,...,parent_package_n;
– use qualifiziert die Komponenten des package_i
use parent_package_1,...,parent_package_n;
package subclass_package **is**
    – eine neue subsorte muss nicht notwendig definiert werden
    – das ist aber notwendig, um universell polymorphe Methoden
    – verwenden zu können
    subtype subsorte_1 **is** sort_1;
        ...
    – Methoden
        ...
end subclass_package;

Instanzierung und Zugriff

Jedem package wird eine Konstruktionsfunktion zugeordnet, z.B. "NewSubsort" und die Instanzierung erfolgt dynamisch:

- dynamisch:
  instref:package.subsort;
  ...
  instref:=NewSubsort(...);
  ...
  instref.component;

Der Zugriff erfolgt wie gewohnt über den Qualifizierer ".", wenn die Subsorte ein strukturierter Datentyp ist, mit einem Index "instref(index)" bei einem Feld und "instref" bei einer skalaren Sorte.

Kommentar

Wir weisen darauf hin, daß die Möglichkeit der Verwendung generischer Einheiten hier noch nicht geprüft wurde. Der Literatur zufolge lassen sich generische Strukturen als Templates verwenden ([Mey89]), etwa

```
generic
  type T is private;
package class_package is
  type classname is something_dependend_of_T;
  methods;
end class_package;
...
package typed_classname is new class_package(type);
...
variable:typed_classname.classname;
```

Ada wird extensiv vom US Militär genutzt. Ada kennt die Polymorhpie als Vorraussetzung für objektorientiertes Programmieren. Ein Klassendefinition im Sinne von Turbo-Pascal oder C++ gibt es in Ada nicht, auch nicht in Ada95. In Ada ist das Bilden von Subtypen möglich. Im Prinzip kann ein package, als Kombination aus einem Datentyp und einer Menge von Funktionen/Prozeduren, als Klasse aufgefaßt werden (siehe auch 2.1.2).
Ada kennt Operatorüberladung (*function "op" (args) return result*).
Unangenehm macht sich das fehlen dynamischer allokierbarer Strukturen resp. Felder bemerkbar.

3.7 Eiffel

Nils Schmeißer

Die Sprache Eiffel wurde 1985 von B. Meyer entwickelt und wird in ihrer aktuellen Version 3 in [Mey92] ausführlich beschrieben.
Eiffel ist eine "rein" objekt-orientierte Sprache, ISE Inc. (Interactive Software Engineering) spricht sogar von der, "(einzig) systematischen Anwendung objektorientierter Prinzipien in allen existierenden Sprachen". Eiffel basiert auf einer kleinen Menge wichtige Konzepte

- Eiffel kann im gesamten Prozeß der Softwareentwicklung eingesetzt werden (Problemanalyse, high-level Softwaredesign, Implementation, ...)
- Klassen verkörpern sowohl Modularität, als auch das Typkonzept. Vererbung steht für Wiederverwendung und Bildung von Subtypen.
- Sorgfältig angelegter und effizienter Zugang zur Mehfachvererbung (renaming, selection, redefinition, undefinition, repeated inheritance).
- Mechanismen zu Schreiben korrekter und robuster Software, zum Debuggen und zur automatischen Dokumentation.
- Ausnahmenbehandlung.
- Statisches Typkonzept (?).
- Dynamisches (=spätes) Binden für Flexibilität und Sicherheit.
- Mechanismen zur Beschreibung flexibler "container structures" (=Templates?).

Für eine komplette Beschreibung der Syntax verweisen wir auf [Swi93], [Wie96] oder [Eif].

Klassendefinition

Die Klassendefinition erfolgt in einer Ada ähnlichen Syntax, die aus dem Bezeichner **class** gefolgt von einer Reihe von Konstruktoren (**creation**) sowie öffentlichen Komponenten in der **feature**-Sektion (man kann auch "**feature** {ALL}" schreiben) besteht.

class classname
creation
    – öffentliche Komponenten
    Konstruktor – ! Deklaration
feature

```
        Attribute
        Methoden
feature {NONE}
        – private Komponenten
end
```

Private Definitionen werden durch das Konstrukt "**feature** {NONE}" eingeleitet.

Vererbung

Soll eine Klasse *classname* erben, so folgt auf die Deklaration "**class** classname" unmittelbar die Angabe der Superklassen mit dem Vorsatz **inherit**.

```
class classname
inherit superclass
creation
...
end
```

Um Namenskonflikte zu Komponenten der Superklassen (Mehrfachvererbung ist möglich) vermeiden zu können bietet Eiffel die Möglichkeit der Umbenennung von Komponenten. Hierfür folgt auf eine Superklasse unmittelbar die Liste der Umbenennungen:

```
class classname
inherit
        superclass rename
                superclass_c1 as neu_c1 ,
                ...
                superclass_cn as neu_cn
        end
end
```

Instanzierung und Zugriff

Eiffel kennt keine statischen Instanzen, d.h. *instvar:classname* deklariert lediglich eine Instanzvariable. Die Instanz selbst wird erst durch den Ruf an den Konstruktor der zugehörigen Klasse erzeugt, also *instvar* **.** *creation_method*.
Der Zugriff auf öffentliche Komponenten erfolgt auch in Eiffel durch das Symbol "**.**".

Kommentar

Eiffel kennt Mehrfachvererbung und Operatorüberladung. Weitere Erfahrungen liegen nicht vor.

3.8 FORTRAN 95/FORTRAN 2000

Ismail Idriss

3.8.1 Introduction

Development and evolution of Fortan

Fortran was one of the first high level languages developed and widely adopted by the academic and scientific community. It was developed over a three year period (1954-1957) by a team at IBM lead by John Backus. Fortran stands for FORmula TRANslation and was used mainly by people with a scientific background for solving problems with a significant arithmetic content.
By 1966 and the first standard it was widely used, easy to teach, had demonstrated the benefits of subroutines and independent compilation, was relatively machine independent and often had very efficient implementations.
By the 1970s it had started to show its age in relation to some of the other languages that had emerged. It was standardised in 1978 (even though the next version was called Fortran 77) and whilst the changes that were made were on the welcome side many felt that they had not gone far enough.
Other language emerged and established themselves. Pascal, Ada, Modula2, C, C++ all became rivals to Fortran in the scientific and academic communities.
Fortran was next standardised in 1991 and is now called Fortran 90. This was a major improvement and the changes are given below.

The new features of Fortran 90

free source form:

- names can be up to 31 characters in length
- blanks are significant
- lines up to 132 characters in length
- up to 39 continuation lines
- ";" statement separator for multiple statements
- "!" as comment symbol
- include option for source text from files

modern control structures:

- Fortran 90 has a modern **DO** statement , with **CYCLE** and **EXIT** options and the control part of the **DO** can be conventional iteration, **WHILE** or no control clause. There is also **CASE** statement specification of numeric precision:

- there is now a clean way to control numeric precision

whole array processing:

- it is now possible to treat arrays as whole objects and write

```
A =  B*SIN(A)
```

  where `A` and `B` are arrays.

dynamic behaviour, including allocate, deallocate, pointers, recursion] user defined data types
modules, and with them come:

- operator overloading
- generic procedure

The new features of Fortran 95

The next revision, Fortran 95, is being published in 1997. It is a relatively minor enhancement of Fortran 90 with a small number of new features and mainly devoted to clarifications, corrections, and interpretations of Fortran 90.
The major new language features in Fortran 95 are:

- FORALL statement and construct,
- Pure and elemental user-defined subprograms,
- Implicit initialisation of derived type objects, and
- Initial association status pointers.

The minor new language features in Fortran 95 are :

- Comments in name list input data,
- Minimal output field width for formatted numeric output,
- Intrinsic SIGN function may distinguish -0 and +0,
- New intrinsic function CPU_TIME returns processor time,
- References to certain pure functions in specification expressions,
- Nested WHERE constructs and masked ELSEWHERE statements,

- Modifications to intrinsic functions CEILING, FLOOR, MAXLOC, and MINLOC,
- Automatic deallocation of allocatable array, and
- Generic identifier in END INTERFACE statement.

The new features of the language bring it up to date and increase the range of problems that can be solved easily. The language provides a very good framework for modern software development.
The separation of the implementation from the design is a very powerful tools for reliable software development. We see in the language the first steps to object orientation with encapsulation and a limited support for polymorphism, without the steep learning curve demanded by C++.

The new features of Fortran 2000

WG5 has determined that Fortran 2000 should be

- A language for high performance numerical, scientific and engineering programming
- A modern language with high quality data abstraction and user extensibility features

The major new features planned for Fortran 2000 are as follows:

- High Performance, Scientific and Engineering Computing:
  - Asynchronous I/O
  - Floating point exception handling
  - Interval arithmetic
- Data Abstraction / User Extensibility:
  - Allocatable components
  - Derived type I/O
  - Object-oriented Fortran :
    - Constructors/destructors
    - Inheritance
    - Polymorphism
    - Procedure pointers
  - Parameterized derived types

- Other Features:
  - Internationalization
  - Interoperability with C

It is the intention of WG5 that the revised standard shall be published no later than November 2002.

3.8.2 Object Oriented Programming in Fortran 2000

One specific defect which hampers both the semantic extension and object oriented paradigms is the lack of the possibility for the user of a type extension module to declare named constants of a type whose structure is private. This essential for true OOP. This problem arises because of the classification of expressions and the restrictions placed on what may appear in initialization expressions.

Constructors and Destructors

We supplement the ALLOCATE statement with a "CONSTRUCTOR" capability and the DEALLOCATE statement with a "DESTRUCTOR" capability. This functionality will bring powerful capabilities to Fortran 2000 with only a small addition in syntax.

The CONSTRUCTOR statement

The CONSTRUCTOR statement defines a subroutine with the ELEMENTAL attribute which is automatically called after the ALLOCATE statement to initialize the allocated quantity (or each component of the allocated quantity if the quantity is an array). The elemental subroutine may have optional arguments which translate into keyword entries of the ALLOCATE statement . The CONSTRUCTOR statement may only appear in a MODULE.
Consider the following example:

```
MODULE OBJ_PACK
     PRIVATE
     PUBLIC  ::  CHAR_OBJ, .....
     TYPE CHAR_OBJ
          PRIVATE :: NTABLE
          CHARACTER(LEN=1), POINTER, DIMENSION( : )  PNEXT
     END TYPE CHAR_OBJ
     INTERFACE CONSTRUCTOR ( )
            MODULE PROCEDURE SUBOO
     END INTERFACE
```

```
    ELEMENTAL SUBROUTINE SUB00(AA, LENGT)
              !  This is an elemental subroutine used as constructor.
        TYPE (CHAR_OBJ) :: LENGT, AA
        IF ( PRESENT(LENGT) ) THEN
               AA%NTABLE=LENGT
               ALLOCATE( AA%PNEXT(LENGT) )
               AA%PNEXT=  '   '
        ELSE
                AA% NTABLE = 0
        END IF
    END  SUBROUTINE  SUB00
    ........
END MODULE OBJ_PACK
```

The DESTRUCTOR statement

The DESTRUCTOR statement defines a subroutine with the ELEMENTAL attribute which is automatically called before the DEALLOCATE statement to clean the quantity (or each component of the allocated quantity if the quantity is an array). The DESTRUCTOR statement may only appear in a MODULE Consider the following example:

```
 MODULE OBJ_PACK
    PRIVATE
    PUBLIC  ::  CHAR_OBJ, .....
    TYPE CHAR_OBJ
         PRIVATE :: NTABLE
         CHARACTER(LEN=1), POINTER, DIMENSION( : )  PNEXT
    END TYPE CHAR_OBJ
    INTERFACE CONSTRUCTOR ( )
           MODULE PROCEDURE SUB00
    END INTERFACE
    INTERFACE DESTRUCTOR ( )
           MODULE PROCEDURE SUB11
    END INTERFACE

    ELEMENTAL SUBROUTINE SUB11(AA)
           !  This is an elemental subroutine used as constructor.
        TYPE (CHAR_OBJ) :: AA
         IF ( AA%NTABLE .GT. 0 )  DLLOCATE(AA)
    END  SUBROUTINE  SUB11
    ........
 END MODULE OBJ_PACK
```

Single Inheritance Module - Type Extension

This provides specifications with illustrative syntax for single inheritance based on the existing derived type mechanism.

Declaring a base type

A type that can be extended shall be declared with the EXTENSIBLE keyword, e. g .

```
TYPE point_2d
     REAL::x,y
   END TYPE
```

This may not be strictly necessary, but ensures object-code compatibility with Fortran with Fortran 90/2000.

Extending a base type

A type can be extended anywhere the base type is accessible. The memory layout of the extended type is such that the base part occurs at the beginning of the type , and components within the base part are laid out the same . it is necessary to supply any new components (the extension type could simply be a re-packing of the original type) but any new component names shall not conflict with accessible component names of the type being extended (the base type). The accessibility of components inherited from the base type is the same as in the base type. An extended type is automatically extensible and need not specify the EXTENSIBLE keyword.

```
TYPE point_3d,EXTENDS TYPE(point_2d)
     REAL::z
   END TYPE
```

It is possible to declare variables of TYPE(point_2d) and those variables are statically typed - the compiler can allocate (the correct amount of) storage and resolve generic references at compile time.
The components of TYPE (point_3d) are x, y and z. It is possible to reference the base, i.e. TYPE(point_2d), part of an entity of this type by using the base part' s type name as component selector. For Example, given:

```
TYPE (point_2d) ::  p2d
TYPE (point_3d) ::  p3d
```

these entities have the components

```
p2d%x, p2%y            ! TYPE(point_2d) only has its own components
p3d%x, p3d%y, p3d%z    ! TYPE(point_3d) has the additional "z" components
```

```
p3d%point_2d          ! plus the base part as a whole ( x and y )
p3d%point_2d%x,
p3d%point_2d%y        ! these are redundant but possible
```

Neither the base type nor the extended type may be a SEQUENCE derived type.

Polymorphic Variables

Polymorphic variables have a declared base type (that may in itself be an extended type) but can contain an object of that type or of any type extended from its declared base type. Therefore they usually cost more to use than normal variables (i.e. those of static - determined at compile-time- type); e.g. they sometimes imply an extra indirection (because of the reference semantics) and sometimes lose optimisation opportunities. They are declared with a separate keyword, e.g.

```
OBJECT (point_2d) :: polly
```

Access via "polly" only provides access to those components in TYPE(point_2d) "polly" can be passed as an actual argument only to a TYPE(point_2d) dummy or an OBJECT(something) dummy where "something " is point _2d or a type extended therefrom. A polymorphic variable can be passed to a dummy argument that is of a parent type. When it is passed to an extended type, the runtime type must be compatible. E.g.

```
CALL sub1(polly)    ! sub1 expects TYPE(point_2d), so this is legal
CALL sub2(polly)    ! sub2 expects OBJECT(point_2d), so legal
CALL sub3(polly)    ! sub3 expects OBJECT(point_3d), legal provided
                    ! polly's runtime type is "point_3d" or extended
                    ! from point_3d.
```

The possible classes of polymorphic variable are:

- Polymorphic dummy arguments. The actual argument can be of any compatible type (as above, this is the declared base type of the dummy argument or any type extended from that base type). An explicit interface is required for a routine with a polymorphic dummy argument.
- Polymorphic pointers. These are polymorphic entities with the POINTER attribute, and may be :
  - local variables
  - functions results
  - structure components

Note that Fortran's auto-dereference facility is ideal for convenient use of polymorphic pointers

For example (*polymorphic dummy arguments*):

```
REAL FUNCTION ARGUMENT ( P )
  OBJECT ( POINT_2D)    ::  P
  argument = ATAN2 ( P%Y, P%X )
END FUNCTION

!  No need to redefine ARGUMENT for POINT_3D objects

REAL FUNCTION AZIMUTH ( P )
  OBJECT ( POINT_3D ) ::  p
  azimuth = ATAN2 ( P%Z, argument(P) )
END  FUNCTION

! P is polymorphic dummy argument so it will work with any later
!  extension of  point_3d,  e.g.  a point_4d .
```

Another example (*polymorphic pointer variables*):

```
OBJECT( point_2d ), POINTER  :: P2
OBJECT( point_3d ), POINTER  :: P3

TYPE( point_2d), TARGET      :: T2
TYPE( point_3d), TARGET      ::  T3

P2 => T2 ; P2 => T3 ; P2 => P3
!  All legal - p2 can point to  a point_2d
!   or any type extended therefrom

P3 => T2   !  llegal, t2 not extended from point_3d

P3 => T3   !  Legal
P3 => P2   !  Legal provided P2 is NULL( ) or its
           !  target is of  TYPE (point_3d) object
           !  or extended therefrom.
```

Note that at this point, by construction, arrays of these "objects" are homogeneous. However, non-homogeneous array-like collections are possible using the same circumlocution which allows arrays of pointers.

Generic Resolution

A polymorphic dummy argument can be a "disambiguator" provided that its Type is "completely incompatible" with that of its corresponding disambiguator. That is, the type of the corresponding argument must be different (kind type parameters) and must not be extended from the same root type. I.e., they must be in different inheritance trees.

Type Enquiry

There are two enquiry functions provided for determining the exact type of a polymorphic variable at runtime:

- SAME_TYPE_AS(POLY,MOLD)
  POLY is a polymorphic variable, i.e. OBJECT (something) POLY MOLD may be polymorphic or of fixed type The result is .TRUE. if POLY is referring to an object of the same actual type as MOLD.
- SIMILAR_TYPE_TO(POLY,MOLD)
  POLY is a polymorphic variable (as above) MOLD may be polymorphic or of fixed type (as above) The result is .TRUE. if POLY is referring to an object of the same actual type as MOLD or of a type extended from that type.

In order for these functions to work each extension type must have a unique signature and polymorphic pointers (and dummy arguments) have a "tag" identifying their type. But there is no need for each subobject to contain this tag - in particular since arrays are homogeneous there is no need for each array element to contain such a tag. It would be natural for this signature to be an internal runtime structure holding type-specific information and for the tag to be the address of this structure.
SAME_TYPE_AS is computable in fixed time simply by the tags on the polymorphic objects. SIMILAR_TYPE_TO can be computed in time proportional to the height of inheritance tree with a small constant space overhead in the type signature, or in fixed time with a space overhead proportional to the height of the inheritance tree.

3.8.3 Procedure Pointer

On further declaration deliberation, including consideration of full committee comments, /data has reached consensus on a declaration syntax for procedure pointer. This syntax avoids the use of "pointer" as an attribute. Instead, the existing "external " attribute is used to distinguish externals from pointer.
Consider the following example:

```
PROGRAM  ptr

     ! -- The abstract interface.
     INTERFACE PROCEDURE ( )
            FUNCTION  real_func(x)
```

```
          REAL, INTENT(IN)  :: x
          REAL :: real_func
     END FUNCTION real_func
     SUBROUTINE sub(x)
           REAL, INTENT(IN) :: x
     END SUBROUTINE sub
END INTERFACE

!-- Some specific external procedures
PROCEDURE(real_func), EXTERNAL :: bessel, gamma
PROCEDURE(sub), EXTERNAL :: print_real

!--  Some procedure pointers, one initialized to null
PROCEDURE(real_func) ::  p, r => null( )  , ptr_to_gamma
PROCEDURE(sub):: s

!-- A derived type with a proc component.
TYPE struct_type
      INTEGER  :: some_int
      PROCEDURE(real_func)  :: component
END TYPE struct_type

!-- and a variable of that type.
TYPE(struct_type) :: struct

!-- Give p a non-null value.
p => bessel

!--Likewise for a structure component
struct%component => bessel

!--Test for equality
IF ( ASSOCIATED(p, struct%component) )    &
     WRITE(*,*) ' this should print.'

!--Evaluate functions
WRITE(*,*)  P(2.5)                          !--bessel(2.5)
WRITE(*,*) struct%component(2.5)            !--r now is gamma

!--pass as an actual argument.
CALL integrate(p, 1.,  2.)

!--Invoke a function returning a proc value.
!--Can we pass gamma directly ?
```

```
    ptr_to_gamma => gamma
    r => select_func( 2, p, ptr_to_gamma)       !--  r now is gamma

    !-- A fairly complicated composition.
    CALL integrate( select_func(1,p,r), 1., 2. )

    !-- Some subroutine operation
    s => print_real
    IF ( ASSOCIATED(s) ) CALL s(3.14)

  CONTAINS

    SUBROUTINE integrate(func, from, to)
       PROCEDURE(real_func), INTENT(IN) :: func
       REAL, INTENT(IN)  ::  from, to

       IF ( .NOT. ASSOCIATED(func) )  CALL abort( 'Oops.')
       WRITE(*,*) ' End values are', func(from), func(to)
          RETURN
    END SUBROUTINE integrate

    FUNCTION select_func(n, proc1, proc2)
         INTEGER, INTENT(IN)  :: n
         PROCEDURE(real_func), INTENT(IN)  :: proc1, proc2
         PROCEDURE(real_func)  ::  select_func

         SELECT CASE(n)
         CASE(1)
                 select_func => proc1
         CASE(2)
                 select_func => proc2
         CASE default
                 select_func => null( )
         END select
         RETURN
    END FUNCTION select_func

END PROGRAM ptr
```

Kapitel 4

OO-Systeme

Nils Schmeißer

Inhalt

In den nachfolgenden Abschnitten werden wir einen kurzen Überblick über Systeme geben, die OO-Mechanismen nuzten. dazu zählen neben grafischen Oberflächen und Betriebssystemen auch Systeme zum verteilten objektorientierten Programmieren und Datenbanken. Auf letztere gehen wir nur sehr kurz ein und verweisen für ein tiefgreifenderes Studium der Materie auf [Heu92].

4.1 Graphische Nutzeroberflächen (GUI)

Die Geschichte der grafischen Nutzeroberflächen beginnt 1972 im Xerox PARC (Paolo Alto Research Center) mit einer, nach neuen Gesichtspunkten (siehe Abb. 4.1) gestalteten grafischen Nutzeroberfläche.
Wesentliches Merkmal einer OO-GUI ist nach [Heu92] die Eigenschaft von Objekten, ihre Einstellung zu behalten und bei erneuter Instanzierung diese zu verwenden.
Betrachten wir zur Zeit verwendete grafische Oberflächen und Betriebssytem(aufsätze) und untersuchen wir diese auf einen OO Hintergrund, so bleiben wir bei

- Smalltalk,
- Oberon
- Next (das System)
- dem GUI des Apple Macintosh,
- dem IBM Presentation Manager für OS/2 und

- *Schreibtisch-Philosophie* mit Dokumenten, Ordnern, Eingangskorb, Papierkorb, etc.
- *generische Kommandos*; Kopieren (drag-and-drop)
- *direkte Manipulation*; Verschieben, Größenänderung
- Darstellung durch *Icons*
- Zuordnung von Eigenschaften an "Objekte", die sich das Objekt auch merkt".

Abbildung 4.1: Gesichtspunkte beim Design der "Alto" Oberfläche (nach [Heu92])

- Microsoft Windows 3.1 und Windows NT
- Turbo-Vision

"hängen". Wir werden gleich ausführlicher über die beiden letztgenannten berichten.
Smalltalk und Oberon haben wir schon vorgestellt, wir wollen an dieser Stelle nur noch einmal auf das Prinzip des Versendens von Nachrichten an Objekte verweisen. Kannte man Oberon bisher als System mit viel Text und wenig grafischen Elementen, so hat sich das im System 3 radikal geändert. Es stehen jetzt grafische Primitive (natürlich auf der Grundlage von Objekten) wie Ikonen und (überlappende) Fenster zur Verfügung.
Kommen wir jetzt zu den MS Produkten. Windows 3.1 ist ein Betriebssystemaufsatz der erstmalig in der MS/DOS-Welt unter dem Stichwort OLE (Object Linking and Embedding) OO-Konzepte verwendet (wir werden weiterhin den Begriff Objekt verwenden, obwohl wir es nicht mit Objekten im Sinne der Definition zu tun haben). Was heißt das? Einem "Objekt" wird die Anwendung (Server), mit der es generiert wurde, zugeordnet. Damit löst der "Aufruf" einer Instanz eines solchen Objektes den Aufruf der zugehörigen Anwendung aus. Ein Objekt kann sowohl eingebettet als auch verknüpft werden. Im letzteren Fall wird die Instanz durch einen Klienten referenziert. Das hat den Vorteil, daß bei einer Änderung der Eigenschaften der Instanz die Änderung in **allen** Klienten sichtbar ist. Beispiel: Eine Excel Tabelle wird sowohl in einem MS-Word als auch in einem Coral-Draw Dokument referenziert. Ändert man die Eigenschaften der Tabelle, so werden die beiden anderen Dokumente sofort aktualisiert (vgl. flaches Kopieren). OLE-2 "täuscht" den Aufruf einer, zum Objekt gehörend Methode dadurch vor, daß der Start der Server-Anwendung transparent ist, d.h. der zum Server-Fenster gehörende Rahmen tritt an die Stelle des Rahmens des Klienten (die vom Klienten stammenden Daten werden davon nicht betroffen)

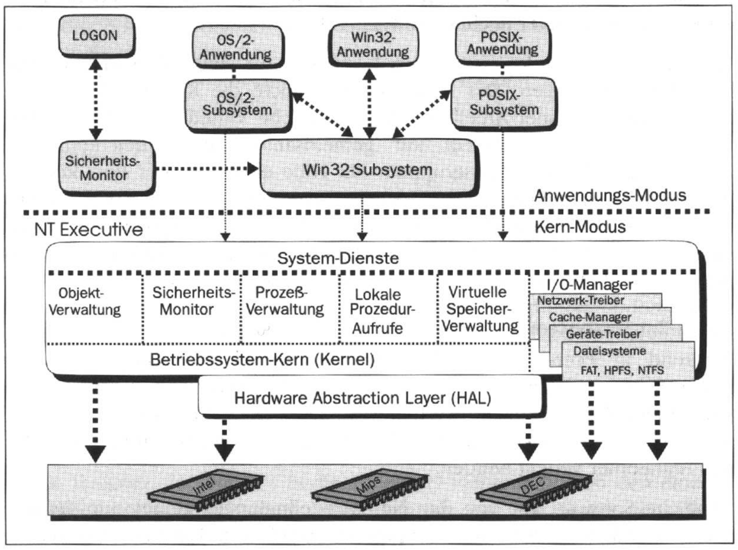


Abbildung 4.2: Der Aufbau von Windows NT (aus [Rod93])

und die verknüpften Daten können mit der Funktionalität des Servers bearbeitet werden. OLE-2 heißt auch, daß eine Verknüpfung durch den Mechanismus drag-and-drop hergestellt werden kann, d.h. die Instanz wird markiert, mit der Maus "gegriffen" und am Ziel "fallengelassen" ([Rod93]). Ab Windows 3.1 können Daten auch direkt Ikonen zugewiesen werden.
Windows NT selbst ist kein objektorientiertes System, es soll aber mit *Cairo* eine echt objektorientierte Oberfläche erhalten. In diesem Zusammenhang möchten wir noch auf die Bemühungen der OMG (Object Management Group) hinweisen, die es sich zum Ziel gemacht hat, eine einheitliche, standardisierte Schnittstelle, über die Objekte miteinander kommunizieren können, zu schaffen (siehe auch 4.2). Windows NT basiert wie das **objektorientierte System Next** auf einem *Mach-Kernel.* Wir verweisen hier nur auf Abbildung 4.2, um den Aufbau von NT zu erläutern.
Das mit "Turbo-Vision" bezeichnete System ist eigentlich keine Oberfläche, sondern eine OO Bibliothek, mit deren Hilfe es möglich ist, eine OO Oberfläche zu schreiben. Bekanntestes Beispiel dafür ist die Borland IDE (Integrated Developers Environment) selbst. Eine neuere Variation zu diesem Thema gibt es als "ObjectWindows" bei Turbo-Pascal 7.0. Wir wollen hier nicht weiter darauf eingehen und verweisen auf [Su93].

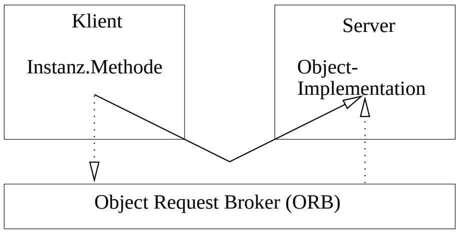


Abbildung 4.3: ORB Konzept

4.2 Verteiltes Objektorientiertes Programmieren

Konzepte des verteilten objektorientierten Programmierens sind zur Zeit noch nicht bzw. nur unvollständig verstanden. Erste Ansätze finden sich in der Common Object Request Broker Architecture (CORBA), in der Distributed Component Architecture (DCOM), dem Pendant von Microsoft sowie in den Java Erweiterungen Java Beans (Sun) und ActiveX (von Microsoft).

4.2.1 Common Object Request Broker Architecture

CORBA ist eine Client-Server Architektur zur Behandlung verteilter Implemenationen, d.h. zu einem Objekt resp. Klasse gehören drei Komponenten

1. die Schnittstelle
2. der Server
3. der Client

Die OMG beschreibt in der OMA (Object Management Architecture), wie ein verteiltes Software System gestaltet werden sollte (im Zentrum steht nicht das Objekt an sich, sondern die (darauf aufbauende) Software). Kernpunkt oder auch grundlegender Baustein der OMA sind Objekte. Im Sinne der OMG sind das alle eindeutig identifizierbaren Einheiten ([Red96]).
Wie funktioniert CORBA? Hinter der Schnittstelle (in IDL[1] formuliert) verbergen sich die, von einem Server angebotenen "Methoden" - das Interface des Servers. Diese Beschreibung wird zunächst übersetzt und daraus Zieltext in einer gängigen OO Sprache (z.B. C++) erzeugt. Der erzeugte Zieltext besteht aus einer Klassendefinition, wobei die zu generierende Klasse **von schon vorhandenen CORBA Klassen** abgeleitet wurde, einem Rahmen für die Implementation eines Servers und eines Klienten. Im Server Rahmen werden die Methoden implementiert, der Klientenrahmen enthält einen vorgefertigten

[1] IDL - Interface Description Language, nicht zu verwechseln mit dem Grafikpaket IDL (Interactive Data Language).

Initialisierungsteil. Im Klienten kann wie gewohnt mit Instanzen umgegangen werden. Der einzige Unterschied besteht darin, daß Anforderungen zur Erzeugung einer Instanz an den ORB bzw. das dahinterstehende System gesendet werden müssen (reference by name).
Vererbung ist im Sinne der Interface-Vererbung möglich, d.h. **die Implementation muß erneut vollständig stattfinden** - CORBA kennt **keine "Implementationsvererbung"**.
Das Prinzip der Polymorphie bleibt gewahrt, d.h. sei $B :: A$, dann kann ein Klient zu A mit dem Servern zu A und zu B zusammenarbeiten, aber ein Klient von B nicht notwendig mit dem Server zu A, [Red96] drückt sich hier nicht klar aus. Es ist zu vermuten, daß ein Klient von B mit einem Server zu A zusammenarbeitet, solange er nur polymorphe Methoden verwendet.
Bevor ein Klient das erste Mal gestartet werden kann, muß der zugehörige Server beim *Implementation Repository* des ORB registriert und gestartet werden. Die Registrierung bleibt i.A. auch nach dem Beenden des Klienten resp. Servers erhalten. Wird erneut ein Klient gestartet, so erkennt das CORBA-System automatisch den benötgten Server und startet diesen. Es kann natürlich mehr als einen Klienten geben (vgl. re-entrant code).
CORBA scheint sich zum Quasi-Standard für objektorientierte Client-Server Architekturen zu entwickeln und wird derzeit u.A. von MPI, OOMPI, HPC++ und PVM unterstützt.

4.2.2 DCOM

DCOM ist wie schon erwähnt, der Microsoft counter part zu den OMG Bemühungen. Obwohl Microsoft angibt, die OMG zu unterstützen, hat die Firma dennoch ihre eigene Beschreibungssprache MIDL entwickelt. Unter die Kategorie DCOM fällt das gesamte OLE-Konzept und ActiveX[2]. Letzteres ist eine Microsoft Erweiterung zu Java, die konkurrente Programmierung ermöglicht.

4.2.3 Java Beans/RMI

Sun schreibt dazu: "Java Beans ist eine architektur- und plattformunabhängige Anwendungsschnittstelle zur Generierung und Nutzung dynamischer Java Komponenenten." Diese Schnittstelle unterstützt insbesondere folgende Funktionalität

- Runtime Component Interface Exposure and Discovery, also die Freigabe der Schnittstellenspezifikation eines Objektes während der Laufzeit
- dito für Attribute eines Objektes
- Ereignisbehandlung

[2]ActiveX ist in letzter Zeit aufgrund entdeckter Sicherheitsmängel und Inkompatibilität zur bestehenden Java-Sprachdefinition und -Semantik in die Schlagzeilen gerückt.

- Persistence, das ist der Mechanismus zur Speicherung von Eigenschaften auf nicht-flüchtigen Speichermedien
- Component Packaging, Kompression von Daten und Programmtext zur schnelleren Übertragung

Für die Verteilung von Anwendungen orientiert sich Java an vorgefertigten und stabil laufenden Systemen wie CORBA.

4.2.4 OOMPI

OOMPI ist eine Bibliothek im OO Stil, die auf MPI aufsitzt, d.h. objektorientiertes Programmieren beschränkt sich hier darauf, MPI Aufruf hinter ein paar Objekten zu verstecken. Beispielsweise wird ein Port-Objekt eingeführt und eine Verbindung durch die Erzeugung zweier Endpunkte=Ports aufgebaut. Eine Kommunikation läuft dann im Prinzip so: 1. Kanal öffnen (zwei Ports einrichten), 2. Daten an Port und damit auf den Kanal senden und 3. Kanal schließen.

Vorname	Nachname	Wohnort	
Maria	Braune	Kleinpösna	
Herbert	Künzel	Dresden	$\Leftarrow$ *Tupel*
Maria	Künzel	Dresden	

$\Leftarrow$ *Attribute*

Abbildung 4.4: Relation Personen - Person "wohnt in" Ort

Vorname	Nachname	Hobby
Maria	Braune	Kurven mit Splines aproximieren
Maria	Braune	Nichtlineare Differentialgleichungen lösen
Maria	Braune	Lernen formaler Sprachen
Herbert	Künzel	Exotische Biere probieren
Maria	Künzel	Häkeln
Maria	Künzel	Sticken

Abbildung 4.5: Relation Hobbys - Person hat Hobby

4.3 Datenbanken

Sören Auer

4.3.1 Relationale Datenbanken

- Menge von Relationen
- Relationen sind Teilmengen des kartesischen Produktes über den Grundmengen der Attribute des Relationenschemas
- Elemente der Relation sind Tupel
- Domänen (Datentypen) für Attribute können meist nur Integer, Real, String oder Boolean sein
- Sprache SQL (' 86, ' 89, ' 93) zur Schemadefinition (Struktur Verknüpfung), Datenmanipulation, Abfrage (Wiedergewinnung) der Information

Anschauliche Interpretation relationaler Datenbanken

- Welt verknüpfter Tabellen (endliche Relationen):
  Attribute $\cong$ Spaltenüberschriften
  Tupel $\cong$ Zeilen
  Einträge der Tabellenfelder enthalten Werte der entsprechenden Domänen (somit nur atomare Attribute werde möglich $\Rightarrow$ erste Normalform)

Generische Operationen (Abfragesprache SQL)

- erlaubt Definition von Relationenschemata, Dateneingabe und -manipulation
- SQL-Anfrage: SELECT Attributliste FROM Relationen WHERE Bedingung
  Beispiel:
  `SELECT * FROM Personen, Hobbys WHERE Wohnort='Dresden'`
  `AND Nachname='Künzel'`

Schwachpunkte

- Trotz Schlüssel, Primär- oder Fremdschlüssel keine Identitäten
- Attribute können nur Realisierungen atomarer Datentypen sein (Auswege - BLOB Binary Large Object, Wertemengen oder Komponenten können simuliert werden $\Rightarrow$ erhebliche Redundanzen oder bei dritter Normalform erheblicher Komplexitätszuwachs)
- Beziehungen zwischen verschiedenen Relationen eines Objektyps, zwischen verschiedenen Objekttypen und Objekt-Komponentenobjekt-Beziehungen können nicht unterschieden werden.

4.3.2 Semantisches Datenbankmodell

Grundlagen für ein semantisches Datenbankmodell bilden Modelle wie:

- Entity-Relationship Modell
- prädikatenlogische Modelle
- mengentheoretische Modelle

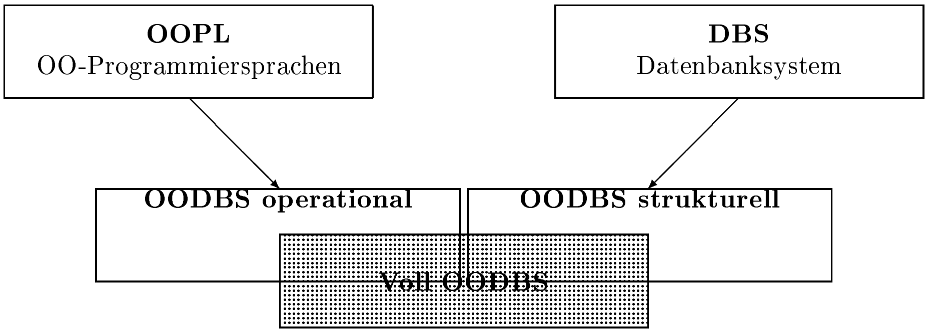
oder

- Modell mit semantischer Hierarchie

Eine ausführliche Erläuterung sowohl zu *SDM* als auch *RDBM* finden sich in [Heu92].

4.3.3 Objektorientierte Datenbanken

Datenmodell legt fest, wie Informationen dargestellt werden und wie darauf zugegriffen wird.



Strukturkonzepte

Objektidentität

Unterscheidung zwischen Objekten und deren Werten (Objekte existieren unabhängig von den Werten ihrer Eigenschaften) $\Rightarrow$ verschiedene Gleichheitsbegriffe

- *Identität* (zwei Objekte sind identisch, wenn sie die gleiche Objektidentität besitzen)
- *flache Gleichheit* (Zwei Objekte sind flach gleich, wenn sie den selben Zustand besitzen.)
- *tiefe Gleichheit* (Zwei Objekte sind tief gleich, wenn sie flach gleich und alle Komponentenobjekte tief gleich sind.)

Es ergeben sich verschiedene Operationen auf Objekten (Tests auf Identität, flache/TiefeGleichheit, flaches/tiefes Kopieren, Zuweisen). Objektidentität wird beispielsweise durch die eineindeutige Zuordnung von *Surrogatwerten* zu Objekten realisiert:

- werden vom System automatisch generiert und gelöscht
- sind global eindeutig
- sind persistent
- dürfen nicht verändert werden

wiederholte Anwendung von Typkonstruktoren; komplexe Objekte

Typenkonstruktoren fassen mehrere Komponenten, Elemente zu einem neuen (komplexen) Typ zusammen:

Konstruktor	Instanz des neuen Typs	Operationen
Tupelkonstruktor ("TUPLE OF")	• Tupel bestehend aus Instanzen der zugrundeliegenden Typen (Komponententypen).	• Komponentenzugriff • Test auf Gelichheit bzw. Ungleichheit
Mengenkonstruktor ("SET OF")	• Menge bestehend aus mehreren Instanzen des zugrundeliegenden Typ (Elementtyp), • keine Ordnung, • keine doppelten Elemente	• Zugriff, Test ('$\in$') auf ein Element • Mengenvergleiche ('=', '$\subset$', '$\subseteq$') • Vereinigung, Durchschnitt, Differenz
Listenkonstruktor ("LIST OF")	• Liste aus Instanzen des zugrundeliegenden Typs (Elementtyp). • Listenelemente sind geordnet • doppelte Elemente möglich	• Zugriff auf erstes nächstes, letztes Element • Durchlaufen der Liste • Konkatenation

Klassen und Typen

Klasse : Menge von Objekten mit gleichen Attributen und Methoden

Typ : strukturelle Beschreibung von (potentiellen) Objekten

Antiisomorphismus (ordnungsumkehrend) zwischen Klassen und Typhierarchie (Galoisbeziehung)

Beziehungen zwischen Klasse und Komponenetenklasse

Gemeinsame (private) Komponentenobjekte :

- Komponentenobjekte können Komponenten mehrerer Objekte (auch unterschiedlicher Klassen) sein (gemeinsame Komponentenobjekte). (Objektstruktur: Graph)
- Komponentenobkekte, welche nur in einem Objekt auftreten sind privat. (Objektstruktur: Baum, Wald)

Abhängige (unabhängige) Komponentenobjekte :

- sind von der Existenz der sie umgebenden zusammengesetzten Objekte abhängig

Eingekapselte (nicht eingekapselte) Komponentenobjekte :

- nur von dem sie umgebenden Objekt aus sichtbar
- können nicht gleichzeitig unabhängig sein

Spezialfälle :

- Referenz: Komponentenobjekte gemeinsam, unabhängig und nicht eingekapselt
- Beziehung: symmetrische Relation zwischen Objekten und deren gemeinsamen, unabhängigen und nicht eingekapselten Komponentenobjekten
- Strukturierte Objekte: Komponentenobjekte privat und abhängig.

Strukturvererbung (Beziehung zwischen Klasse und Unterklasse)

Klassenhierarchie : Integritätsbedingung an die beteiligten Objektmengen: Die Unterklasse hat weniger Objekte als die Oberklasse.

Typenhierarchie : Vererbung des Verhaltens von Objekten: Die Unterklasse hat mehr Attribute als die Oberklasse (erbt die Attribute der Oberklasse).

Stimmen Klassen- und Typhierarchie überein $\Rightarrow$ Is-A-Hierarchie.

Integritätsbedingungen

sind weitestgehend überflüssig geworden, da inhärent in OODM-Konzepten enthalten.

Schlüssel identifizieren ein Objekt (sinnvoll, da Identitätsidentifikator für Benutzer nicht sichtbar). Möglich sind auch vererbte, komplexe und Komponentenobjektschlüssel.

Kardinalitäten legen bei mengenwertigen Attributen die Mindest- oder Maximalanzahl der Elemente fest.

Überdeckungsbedingung an die Strukturhierarchie: Vereinigung der zu den Unterklassen gehörenden Objektmengen muß mit der Objektmenge der Oberklassen übereinstimmen.

Disjunktheitsbedingung : ein Objekt einer Klasse kann maximal in einer Unterklasse auftreten.

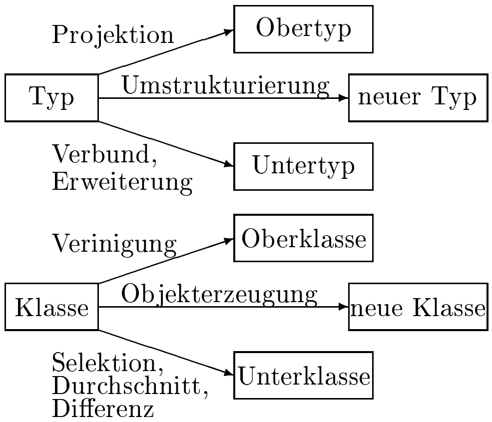


Abbildung 4.6: generische Operationen für OODMs

Operationen

Relationale Operationen

- Objektmengen werden als geschachtelte Relationen betrachtet
- unter Verzicht auf die Berücksichtigung der Objektidentität, des Klassenkonzeptes und der Strukturhierarchie sind Projektionen, Selektion, Mengenoperationen und Verbünde möglich.

Objekterzeugende Operationen

- Ergebnis objekterzeugender Operationen sind neue Objekte mit den durch die Operation spezifizierten Eigenschaften.
- Objektidentität wird nicht bewahrt

Objekterhaltende Operationen

- berücksichtigt Objektidentität
- ermöglichen verschiedene Sichten, dynamische Klassifizierung/Einkapselung

Unterstützung einer Abfragesprache / Programmiersprachenanbindung

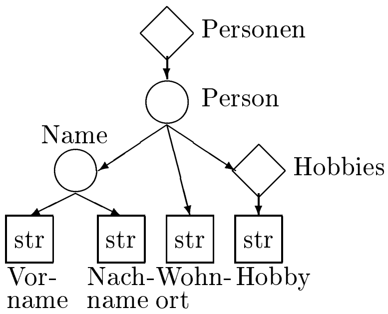
- Methoden können in objektorientierten Programmiersprachen geschrieben werden
- Abfragesprachen: SQL, ODBC, OQL (Obermenge von SQL 2.0)

Höhere Konzepte

Metaklassen, Definition- und Vererbung von Methoden, Abstrakte Datentypen und Kapselung, Overriding und Mehrfachvererbung

Beispiel

```
SET OF (
   TUPLE OF (
      Name: TUPLE OF (
         Vorname: STRING,
         Nachname: STRING),
      Wohnort: STRING,
      Hobbys: SET OF (
         Hobby: STRING)
   )
).
```



Konzepte objektorientierter Datenbanksysteme

[Heu92] gibt hierfür folgende Konzepte an:

Erweiterbarkeit : Erweiterbarkeit eines Systems auf allen Ebenen um Typen, Funktionen und Speicherstrukturen

Persistenz : Fähigkeit des Systems seinen Zuständ transparent auf ein nicht-flüchtiges Speichermedium zu übertragen und von dort zu rekonstruieren

Transaktion : Fähigkeit des Systems, Transaktionen aus elementaren Operationen auf der Objektbank zu bilden

Concurrency Control : Fähigkeit, parallele Transaktionen zu synchronisieren

Recovery : Fähigkeit der Anwendung, nach einem Systemfehler einen letzten konsistenten Zustand zu rekonstruieren

Als kommerzielle Beispiele für OODBS werden in der selben Quelle GemStone, ONTOS/VBASE, O_2, ITASCA/ORION und ObjectStore aufgeführt, als freie Systeme POET und POSTGRESS.

Anhang A

Nutzung der Systeme

N. Schmeißer

Inhalt

In diesem abschließenden Kapitel wollen wir den Umgang mit den, an die behandelten Programmiersprachen gekoppelten Programmiersystemen erläutern. Wir beziehen uns dabei sowohl auf kommerzielle Produkte (Turbo-Pascal von Borland, Eiffel von SIG Computer GmbH) als auch auf frei Produkte. Alle erläuterten sind auch als freie Implementationen verfügbar.

A.1 Turbo-Pascal

Turbo-Pascal ist ein kommerzielles System, das zu einem Paket mit der *IDE* - Integrated Developers Environment, einem stand-alone Debugger, einem Profiler, diversen Bibliotheken und einem Linker zusammengefaßt wurde. Die IDE ist die bevorzugte Entwicklungsumgebung (der Compiler selbst ist auch als Kommandozeilenversion vorhanden). Die IDE wird mit "turbo.exe" (DOS Real-Mode) oder "bp.exe" (DOS Protected-Mode) gestartet (eine Windows-IDE ist ab Version 7.0 ebenfalls verfügbar, aus der Sicht des Autors ist aber dringends von der Benutztung abzuraten, so lange keine MS-Windows Programme erstellt werden sollen). Es erscheint jetzt eine, sich am SAA-Standard [1] orientierende Oberfläche. In der IDE ist ein Editor vorhanden, in welchem die Quelltexte eingegeben werden. Über ein Menü bzw. über Tastenkombinationen erlangt man neben Dateioperationen auch Zugang zum Compiler. Mit "Compile" (Alt-F9) wird der Inhalt des aktuellen Fensters übersetzt, F9 - "Build All" übersetzt das gesamte

[1] SAA - System Application Architecture, das ist ein Standard, der vorschreibt, wie ein Fenstersystem auszusehen hat und an den sich kaum jemand hält (siehe Windows 95 oder Sun/OpenLook)

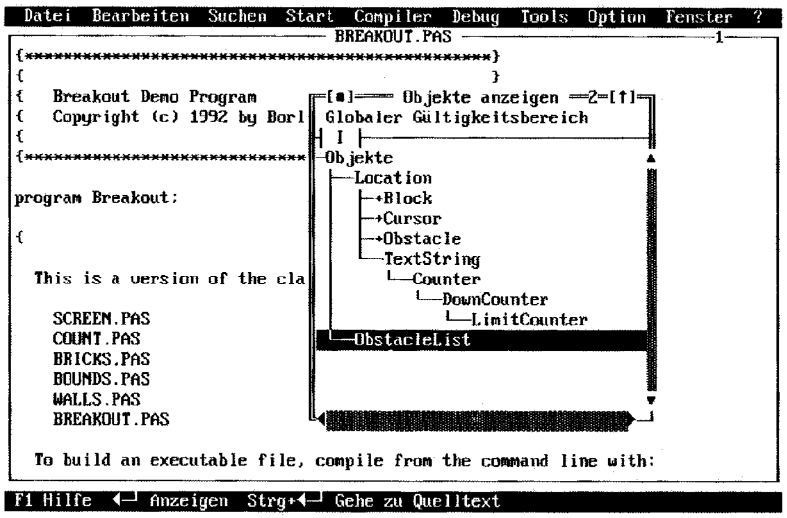


Abbildung A.1: Borland DOS-IDE

Program, einschließlich aller Units. CTRL-F9 startet das Programm. Das Starten einer Applikation in der IDE gibt uns die Möglichkeit, Fehler im Programm zu erkennen und zu finden. Hierfür zeichnet der interne Debugger verantwortlich, der in der IDE durch Setzten von Unterbrechugspunkten, schrittweiser Programmabarbeitung usw. gesteuert werden kann. Ein ausführliches, kontext-sensitives Hilfesystem ist vorhanden.
Alternativ kann der GNU Pascal-Compiler verwendet werden. Er ist frei, aber an den GNU C-Compiler gebunden. Mit dem GPC konnten bisher noch keine Erfahrungen gesammelt werden.

Verfügbarkeit : Turbo-Pascal ist im PC-Pool ab Version 6.0 vorhanden (MS-DOS), für Linux gibt es das GNU-Packet "gpc".

A.2 Oberon

Oberon ist ein freies System, das direkt vom FTP-Server der ETH Zürich bezogen werden kann.
Nach Installation wird Oberon durch den Aufruf "oberon" gestartet. Auf der rechten Seite ist i.A. ein *Viewer* mit einigen wichtigen Methoden des Systems zu finden. Wir schaffen uns zunächst durch Klicken mit der mittleren Maustaste von "Edit.Open" einen freien Viewer, in welchem wir weitere Anweisungen plazieren können. Dazu plazieren wir das "Carret" durch Klicken mit der linken Maustaste in den freien Viewer und senden die Nachricht "Edit.Open

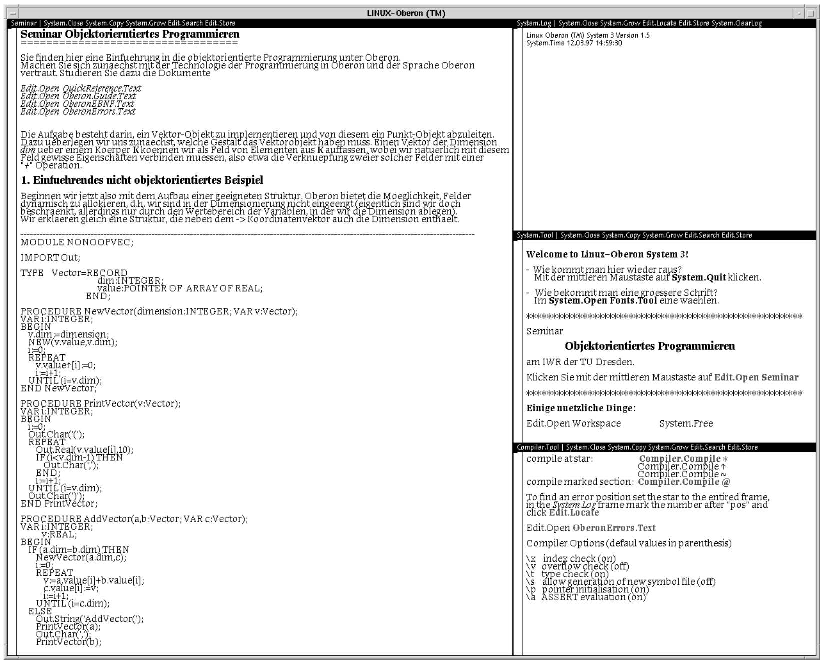


Abbildung A.2: Oberon Oberfläche

	nur eine Taste	ML	MM	MR
ML	Einfügemarke setzen (star)	-	kopieren (hierher)	Attribut kopieren
MM	Ausführen	free & execute	-	Bildschirm-schoner
MR	Markieren	Löschen	Kopieren (dorthin)	-

Abbildung A.3: Belegungen der Maustasten unter Oberon, die Zeile gibt die zuerst gedrückte Taste an

QuickReference.Text" wiederum durch Klicken mit der mittleren Maustaste auf "Edit.Open" an das System. Sollte "Edit.Open" zum Öffnen eines Viewers nirgends zu finden sein, so wird das Carret einfach an eine frei Stelle positioniert und über die Tastatur der Text "Edit.Open" eingegeben. Oberon macht extensiven Gerbauch von der Maus. In Tabelle A.3 sind alle wichtigen Kombinationen zusammengefaßt.

Verfügbarkeit : vom FTP Server der ETH (ftp://ftp.inf.ethz.ch/pub/Oberon/) für Amiga, DECstation, HP700, Linux, MacII, PowerMac, RS6000, SPARC, SiliconGraphics.

A.3 Smalltalk-80

Smalltalk ist neben kommerziellen Versionen auch als freie (für Ausbildungszwecke) Software für Linux und frei als Beta-Release für MS-Windows verfügbar. Nach dem Start mit "smalltalk" erscheinen i.A. vier Fenster ("smallTalk", "Transcript", "System Browser" und "Workspace"). Im Workspace kann, genau wie in einem Viewer von Oberon, eine Nachricht spezifiziert werden. Bei Smalltalk muß die Nachricht zunächst markiert werden (linke Maustaste Klicken und Halten). Danach wird mit der mittleren Maustaste (immer noch im Workspace) ein Pulldown-Menü geöffent und die markierte Nachricht mittels "doIt" oder "printIt" versendet. Mittels des "FileBrowser" können Dateien importiert werden. Der Browser läßt sich alternativ durch Versenden der Nachricht "FileBrowser start." aufrufen (den Punkt beachten). Nach Auswahl einer Datei (Doppelklick mit der linken Maustaste) kann diese mit "fileIn" in das System geladen werden (fileIn im Pulldown-Menü der mittleren Maustaste). Scripte können auch direkt im Anzeigebereich des File Browsers wie im Workspace ausgeführt werden.
Mehr Informationen sind über das "smallTalk" Menü erhältlich.

Verfügbarkeit : Smalltalk kann für MS-Windows über "http://www.intuitive.co.uk/dolphin/dolphin.htm". Eine, für Lehr- und Forschungszwecke freie Implementation steht unter Linux zur Verfügung.

Abbildung A.4: Smalltalk Oberfläche

A.4 C++

C++ gibt es von Borland wieder in einer IDE-Version (Aufbau ähnelt dem der Turbo-Pascal IDE). Ansonsten ist es eher üblich, C/C++ als Kommandozeilencompiler zu betreiben. Eine wesentliche Rolle bei der Erstellung von Programmen mit C/C++ spielt das "make" Werkzeug (in der B.-IDE: "Project"). Ein Makefile gibt an wie einzelne Komponenten des Programmes (Module) voneinander abhängen. Daraus wird dann die Reihenfolge der Übersetzung bestimmt und das Ziel "zusammengebaut". Bevor wir zu einer kurzen Erläuterung des Makefiles schreiten noch ein Wort zur Borland IDE: Nach Öffnen eines neues Projektes werden diesem einfach mitgeteilt, welche Dateien (Modulquelltexte **und Bibliotheken**) dazu gehören. Die IDE kennt das Kommando "Make" bzw. F9 und initiiert den Zusammenbau.
Auf UNIX Systemen gibt es keine IDE[2], deshalb wird hier, wie schon erwähnt vorrangig "make" verwendet. Natürlich können Programme auch per Hand zusammengebaut werden, davon ist aber (zumindest bei größeren Projekten) dringendst abzuraten. Das, den ganzen Vorgang steuernde "Makefile" (übliche Namen sind "Makefile" oder "makefile") enthält zwei Abschnitte; einen Definitionsteil für Variablen und einen Regelteil. Variablen (genauer Konstanten) werden gemäß

```
variable_definition ::= variable_name "=" value
```

festgelegt. Auf Variablen kann sofort nach ihrer Definition mit "$(variable_name)" zugegriffen werden. Der Regelteil besteht aus einer Abfolge von Regeln, die auf der linken Seite ein Ziel angeben, abgetrennt durch einen Doppelpunkt gefolgt von einem Tabulator stehen die Dinge, von denen das Ziel abhängig ist. Auf der nächsten Zeile, wiederum mit einem Tabulator vom Zeilenanfang abgesetzt wird erklärt, wie das Ziel erzeugt wird.

```
rule ::= target ":" [ "\t" dependencies ] [ "\n" "\t" action ] "\n"
dependencies ::= filename [ dependencies ]
action ::= executable_instruction
```

Beispiel:

```
# der IBM AIX C++ Compiler
CC = xlC
# soll auch Dateien, die auf .c enden als C++ Quelltext akzeptieren
CFLAGS = -+
# Optionen fuer den Linker: keine
```

[2]Das stimmt nicht ganz: unter Linux gibt es "wpe" bzw. "xwpe", eine, der Borland-IDE nachempfundene Oberfläche. Die bisher damit gemachten und in news-groups disskutierten Erfahrungen scheinen allerdings darauf hinzudeuten, daß die Verwendung des "make"-Werkzeuges sinnvoller ist.

```
LFLAGS =
# Bibliotheken: z.B. X11 Bibliotheken  libX11.a, libXt.a
# die Angabe des Bibliothekspfades wuerde man normalerweise
# den LFLAGS zuordnen
LIBS   = -L/usr/lib/X11 -lX11 -lXt

#      v---- Tabulator
xziel:   main.o modul1.o modul2.o
         $(CC) -o xziel $(LFLAGS) main.o modul1.o modul2.o \
         $(LIBS)
# ^----- auch hier Tabulator, Zeilen koennen mit einem \ umgebrochen
#        und auf der naechsten fortgesetzt werden (nach dem \ MUSS
#        mindestens ein Terminator stehen, z.B. Leerzeichen oder Tab

main.o:  main.c modul1.h modul2.h
         $(CC) -c $(CFLAGS) main.c

# es gibt auch generelle Regeln
.SUFFIXES: .cc

.cc.o:
         $(CC) -c $(CFLAGS) $<
```

Bemerkung die generellen Regeln sind bei einer größeren Anzahl von Modulen sinnvoll, die alle nach dem selben Schema erzeugt werden.
Der Vorgang wird einfach durch den Aufruf von "make" gestartet. Sind die Regeln in einer anderen Datei als "Makefile" angegeben, so wird "make -f fname" aufgerufen.

Verfügbarkeit : Als kommerzielles Produkt von Borland (Borland C++ $\geq$ V 2.0), von Microsoft (z.B. Visual C++) und diversen anderen Herstellern. Als freie Software (GNU) gibt es den gcc/g++ auf verschiedenen FTP-Servern für so gut wie alle Plattformen. Der gcc wird meist als Quelltext geliefert, er gehört zum Umfang von Linux. Für MS-DOS ist er unter "http://www.it.dtu.dk/ jan/4361/Tools/djgpp/" zu finden.

A.5 Java

Der Java-Compiler wird zusammen mit dem Java Development Kit (JDK) von Sun **frei** weitergegeben. Nach Installation steht i.A. der Compiler (javac) und eine Applikation zum Testen der erstellten Applets (java) zur Verfügung. Applets können auch mit einem entsprechenden WWW- Browser (Netscape) getestet werden. Werden Module zu Kategorien zusammengefaßt (vgl. Smalltalk), so

werden diese in einem Unterverzeichnis mit dem Namen der Kategorie plaziert. Der Java-Compiler erkennt selbstständig Abhängigkeiten.
Zum Testen kann "java appletname" (ohne die ".class" Erweiterung) verwendet werden.

Verfügbarkeit : Das JDK gibt es direkt bei Sun unter "http://java.sun.com/" für Sun/Solaris und Windows95/NT. JDK ist seit einiger Zeit Bestandteil der Linuxdistributionen. Linux kann ab Kernel-Version 2.x auch Applets direkt ausführen (es ist keine virtuelle Maschine mehr nötig).

A.6 Ada 95

Auch für Ada gibt es neben kommerziellen Versionen eine freie, das "Gnat" Projekt (GNU NYU Ada 9X Translator). gnat arbeitet nur zusammen mit dem GNU C Compiler. Der Zusammenbau eines Programmes wird nach Installation von gcc und gnat am besten wieder mit "make" gesteuert. Die einzelnen Module werden mittels "gcc" übersetzt (nur "-c", also nur Erzeugung von .o Dateien) und anschließend mit "gnatbl" gebunden. Bei der Übersetzung werden .ali Dateien erzeugt, die alle, für gnatbl nötigen Informationen zum Zusammenbau der Applikation enthalten.

Verfügbarkeit : Gnat gibt es für OS/2, HP PA (Hewlett-Packard), Sparc Solaris, Sun OS, DEC Alpha, SGI Irix, Linux (ELF), PowerMac, Windows95/NT, zu finden unter "ftp://cs.nyu.edu/pub/gnat/".

A.7 Eiffel

Eiffel wird von der SIG Computer GmbH für verschiedene Plattformen vertrieben, u.A. für MS-DOS, MS-Windows und Linux. Für den Betrieb ist ein C-Compiler nötig, da Eiffelcode zunächst in C umgesetzt wird. Die Programmgenerierung wird bei Eiffel über eine ".pdl" Datei gesteuert. In dieser Datei wird der Name der Applikation, die "Wurzelklasse" mit Konstruktor und Cluster angegeben. Von der Wurzelklasse wird eine Instanz erzeugt und deren Konstruktor gerufen (vgl. *main* in C/C++). Ein Cluster in der Eiffel Terminologie ist eine Menge zusammengehöriger Klassen. Da eine Klasse immer in einzelnen Dateien abgelegt wird, ist es nur natürlich, das ein Cluster mit einem Verzeichnis verknüpft ist[Eif].

```
program MyApp

root
    MyApp : "make"
```

```
cluster

    mycluster:"./"
    end

include "$EIFFEL_S/library/lib.pdl"
include "$EIFFEL_S/library/lib.lib"

end
```

Verfügbarkeit : Eiffel Compiler für MS-DOS und MS-Windows sind unter "http://www.informatik.th-darmstadt.de/mitarbusers/~boch/eiffel/bin/" zu finden. Eine frei Demo-Version für Linux findet sich auf "sunsite.unc.edu" oder einem Mirror unter "pub/Linux/devel/lang/eiffel" (Limitierung auf 100 Klassen, sonst volle Funktionalität).

Anhang B

Übungen

Nils Schmeißer

Inhalt

Die nachfolgenden Aufgaben stellen ein Sammlung von Standardproblemen des objektorientierten Programmierens dar. Um dem Leser eine Anregung zum Einstieg in das objektorientierte Programmieren zu geben, empfehlen wir die Aufgaben B.1 bis B.2 vollständig abzuarbeiten. B.3 bis B.6 stellen größere Programmierprojekte dar, die aber durchaus mit einem Aufwand $\leq$ ein Wochenende gelößt werden können.

B.1 Euklidische Geometrie

Das 1899 von David Hilbert (1862-1943) komplettierte Axiomsystem der Euklidischen Geometrie (Euklid, 365-300 v.u.Z.) stellt die Grundlage für viele praktische Anwendungen der Geomtrie dar.

Definieren Sie ausgehend von dieser Beschreibung Objekte und Basisklassen für Primitive der ebenen euklidischen Geometrie, etwa Punkt, Strecke, Strahl, Gerade, Fläche, Winkel, Kreis, (geradlinige) Figur. Erklären Sie Methoden mit denen Sie Grundoperationen wie Erzeugung, Schnitt (etwa Gerade $\cap$ Gerade $= \{$ Punkt,Winkel $\}$) und Vereinigung ausführen können. Verwenden Sie eine, Ihnen genehme, verständliche formale Notation.

Ziel : Übergang von formalen "natürlichen" Definitionen zur Beschreibung im OO Stil (Attribute und Methoden); Verständnis der Begriffe Klasse, Objekt, Instanz, Attribut, Methode; Vererbung

B.2 Verfahren und Methoden der Algebra

B.2.1 Algebraische Strukturen (Mengen, Gruppen, Ringe, Körper)

Definieren Sie ein System von Klassen/Objekten, mit denen Sie Mengen, algebraische Strukturen, Gruppen, Ringe und Körper erfassen können. Verwenden Sie eine Programmiersprache Ihrer Wahl.
Üben Sie sich im Umgang mit Instanzen; Erzeugen Sie z.B. eine Menge und führen Sie Operationen aus (Schnitt, Vereinigung, ...). Definieren Sie einen Restklassenring Modulo n und rechnen Sie darin.

Ziel : Vererbung, Instanzierung, Operatoren, Auswahl einer bekannten OO-Sprache

B.2.2 Vektoren und Matrizen

Definieren Sie zwei Templates mit denen Sie Vektorräume und Matrizen über beliebigen Körpern erklären können. Verwenden Sie ihre Körperdefinition aus der vorhergehenden Aufgabe.
Machen Sie sich bei der Lösung dieser Aufgabe auch mit dem modularen Programmierstil unter C++, Turbo-Pascal und Java vertraut.
Definieren Sie die selben Klassen für dünnbesetzte Vektoren/Matrizen.

Ziel : Templates, Vererbung, Wiederverwendung von Code

B.2.3 Taylorarithmetik

Eine Abbildung $f : D \to W$, die in $(x_0 - \alpha, x_0 + \alpha)$ $(\alpha > 0)$ $(n+1)$-mal stetig differenzierbar ist, läßt sich gemäß dem Satz von Taylor in eine Reihe

$$f(x) = \sum_{i=0}^{n} \frac{f^{(i)}(x_0)}{i!}(x - x_0)^i + R_n(x) \tag{B.1}$$

mit

$$R_n(x) = \frac{f^{(n+1)}(x_0 + \theta(x - x_0))}{(n+1)!}(x - x_0)^{n+1}, \theta \in (0,1) \tag{B.2}$$

entwickeln.
Definieren Sie eine Menge aus Elementen "Taylor", die eine, nach n Gliedern abgebrochene Taylorreihenentwicklung darstellen (n sei für alle Elemente der Menge gleich), wobei das Restglied vernachlässigt werden soll. Erklären Sie in dieser Menge alle gägigen Operationen $(+, -, *, /)$. Erklären Sie weiterhin die n-te Ableitung und eine Menge von ebenfalls gängigen Funktionen in dem Sinne, sei $T_n(f)$ die Taylorreihenentwicklung und existiere die Reihenentwicklung von g(f), dann sei $g(T_n(f)) = T_n(g(f))$.

Ziel : Operatorüberladung, Funktionsüberladung (das ist eigentlich eine nicht OO spezifische Eigenschaft bestimmter Sprachen)

B.3 Formelmanipulation

Entwerfen Sie ein "Framework" für ein Formelmanipulationssystem, mit welchem Sie Ausdrücke verknüpfen und evaluieren können, also z.B. $A_1 = x + y * 2$, $A_2 = z - 4 * x \rightarrow A_3 = A_1 + A_2 = -3 * x + y * 2 + z$.
Kernstück sollte ein Ausdruck-Objekt sein, daß einen String akzeptiert und in eine interne Darstellung umsetzt (Berechnungsgraph aus Ausdruck-Primitiven). Mehrere solcher Objekte sollten sich durch Operatoren zu einem neuen Ausdruck vernküpfen lassen. Definieren Sie eine Methode zur Evaluierung eines Ausdruckes und zur Ausgabe.
An die Lösung dieser Aufgabe kann man mit einer bottom-up oder einer topdown Strategie herangehen. Verfolgen Sie eine dieser Strategien programmtechnisch und geben Sie die andere in formaler Form an.

Ziel : Definition abstrakter Basisklassen, virtuelle Methoden, Aufbau von Bäumen, Traversierung

B.4 attributierte Graphen

Motivation: Ein Bild enthält verschiedene grafische Elemente, die auf eine bestimmte Art und Weise angeordnet sind (Syntax) und damit eine bestimmte Information repräsentieren (Semantik). In diesem Sinne stellt ein Bild ein Wort einer Sprache dar. Um ein gegebenes Wort (Bild) zu verstehen, muss es u.U. in eine andere Sprache übersetzt werden. Beispiel: Wenn wir an einer Fußgängerampel stehen so sehen wir die Ampel, erkennen, welches der beiden Lichter aktiv ist und können mittels der uns bekannten Verkehrsregel schlußfolgern, ob wir die Straße überqueren dürfen oder warten müssen. Diesen Entscheidungsprozeß könnte man etwa wie folgt beschreiben:

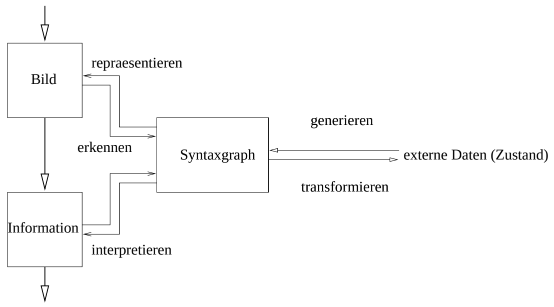
1. Suche im Bild nach bekannten Strukturen (Ampel [1]) = syntaktische Analyse

2. Erfrage den Status der Ampel (rot bzw. grün) = semantische Analyse

3. Assoziiere den Status mit der Aktion $Warten = rot$ bzw. *Gehen* = *grün* = Synthese

[1] Beachten Sie hierbei, daß eine Ampel wiederum aus verschiedenen Komponenten besteht (rotes und grünes Licht), die sich exklusiv in einem bestimmten Zustand befinden, der den Zustand der Ampel bestimmt.

Um den zweiten Schritt ausführen zu können, ist es nötig, die extrahierten Informationen in einer geeigneten Struktur abzulegen, etwa in einem Baum, dessen Wurzelknoten die Ampel und dessen Blätter das rote und das grüne Licht sind und von denen eins exklusiv als aktiv markiert wird (attributierter Graph). Der zweite Schritt wird auch Traversierung genannt und läuft nach fest vorgegebenen Regeln ab, also etwa $RotesLicht.Status = aktiv \rightarrow Ampel.Status = ROT \vee GrünesLicht.Status = aktiv \rightarrow Ampel.Status = GRÜN$. Folgendes Schema gibt die Zusammenhänge wieder.



Gegeben sei folgende Situation: Sie befinden sich an einer Kreuzung mit vier Einmündungen. An jeder Einmündung befindet sich eine Verkehrsampel mit rotem, gelben und grünem Licht.

1. Programmieren Sie eine Zeitsteuerung für die Ampeln und einen "Bildgenerator", der auf Anfrage eines eintreffenden Fahrzeuges das, für dieses Fahrzeug sichtbare "Bild" generiert (= generieren Sie eine Zeichenkette "***OO**" für eine Ampel mit rotem Licht, "****O**" für rot-gelb, "**OO***" für grün, und "**O*O**" für gelb.

2. (a) Entwerfen Sie eine Objekt-/Klassenhierarchie mit deren Hilfe Sie den, zur Analyse des Bildes notwendigen attributierten Graphen aufbauen können.

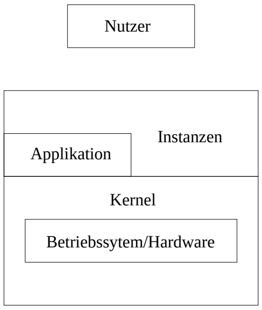
   (b) Versehen Sie den Graphen resp. die Knoten des Graphen mit einer Methode zur Traversierung. Bedenken Sie dabei, das Sie eventuell eine Traversierung in einem Subgraphen ausführen müssen.

3. Schreiben Sie ein Programm zur Simulation des Verkehrsflusses an einer solchen Kreuzung. Gestalten Sie das Programm skalierbar, also etwa variabel in der Anzahl der Einmündungen ($2 \leq n \leq 6$, der Fall $n = 2$ charakterisiert eine Fußgängerampel). Verwenden Sie statt eines einzelnen Fahrzeuges an der Ampel eine Warteschlange.

Ziel : virtuelle Methoden, Verknüpfung verschiedener Objekttypen

B.5 Grafisches Basissystem

Überlegen Sie sich, welche Elemente ein Graphical User Interface (GUI) benötigt und welche Eigenschaften diese besitzen müssen. Geben Sie sinnvolle Verwandschaftsbeziehungen an. Entwerfen Sie eine entsprechende Objekt-/Klassenhierarchie. Geben Sie ein Kernel [2] an. Sie können sich beim Kernel auf die Eingabe von Tasten bzw. ganzen Kommandos beschränken (geben Sie etwa einen Prompt aus und lassen Sie vom Kernel Befehle wie "klickat x,y" akzeptieren).



Ziel : Definition abstrakter Basisklassen, virtuelle Methoden, dynamische Instanzierung, Nachrichten-Paradigma, Ereignisbehandlung

B.6 minimales Datenbanksystem

Generieren Sie eine Objekt-/Klassenhierarchie zur Verwaltung von Bäumen/Graphen und benutzen Sie dieses, um eine kleine Datenbank zu schreiben, welche das Hinzufügen, Löschen und binäre Suchen von Datensätzen ermöglicht.

Ziel : Effiziente Verwaltung einer großen Anzahl von Instanzen, Graphen, Kobination von OOP mit herkömmlicher Programmierung

[2] Ein Kernel stellt die Schnittstelle zwischen der Applikationsschicht und dem Betriebssystem dar. Die Funktionalität des Systems kann nur über das Kernel benutzt werden (Transparenz). Das Kernel empfängt Anfragen des Systems, generiert daraus Ereignisse (z.B. Maustaste gedrückt) und sendet diese an **alle**, ihm sichtbaren Instanzen, die dann ihrerseits selbst für die Behandlung zuständig sind. Umgekehrt empfängt das Kernel Nachrichten von Instanzen und setzt diese in Systemaufrufe um (z.B. Ausgabe einer Zeichenkette an einer bestimmten Position oder Zeichnen eines Rechteckes). Das Kernel kann im Polling-Modus arbeiten (eine Schleife fragt explizit das Betriebssystem nach Anforderungen ab) oder im Interruptbetrieb (das System unterbricht einen gerade laufenden Prozeß und ruft eine ereignisgenerierende Kernelfunktion auf, nachdem das Kernel die Anforderung bearbeitet hat, also ein Ereignis generiert und an die Instanzen versendet hat, übergibt das System die Kontrolle wieder an den unterbrochenen Prozeß.

B.7 Beispiele

Anhang C

Beispiele

Nils Schmeißer

Inhalt

Wir haben Lösungen folgender Aufgabe in allen behandelten Programmiersprachen implementiert:
Für einen Koordinatenvektor der Dimension n sollen die Grundoperationen *Zuweisung*, $+$, $-$ und Skalarprodukt erklärt werden. Das Setzten und auslesen der Komponenten soll über einen Satz von Methoden ermöglicht werden. Es soll eine Methode zur Ausgabe des Koordinatenvektors auf dem Bildschirm vorhanden sein.
Von diesem allgemeinen Fall soll eine spezielle Klasse "2-dimensinaler Punkt" mit spezifischen Methoden abgeleitet werden.

C.1 Turbo-Pascal

MyApp.pas

```
program linalg;

const MAXDIM=256;

type pvector=^vector;
     vector=object
              dim:integer;
              x:array[1..MAXDIM] of real;

              constructor Init(d:integer);
              function GetV(i:integer):real;
```

```
              procedure SetV(i:integer; v:real);
              function GetDim:integer;

              { addiere a zu b und gib Zeiger auf neue Instanz zurueck,
                das ist problematisch, da Pascal keine garbage-collection
                kennt }
              function Add(v:vector):PVector; virtual;
              function Sub(v:vector):PVector; virtual;
              function Scalar(v:vector):real;
              procedure Print; virtual;
            end;

     PPoint=^Point;
     Point=object(Vector)
             constructor Init;
             procedure SetX(v:real);
             procedure SetY(v:real);
             function GetX:real;
             function GetY:real;
           end;

constructor Vector.Init(d:integer);
begin
  dim:=d;
end;

function Vector.GetV(i:integer):real;
begin
  if (i>=1) AND (i<=dim) then
    GetV:=x[i]
  else
    GetV:=0;
end;

procedure Vector.SetV(i:integer; v:real);
begin
  if (i>=1) AND (i<=dim) then
    x[i]:=v;
end;

function Vector.GetDim:integer;
begin
  GetDim:=dim;
end;
```

```
function Vector.Add(v:vector):PVector;
var p:pvector;
    i:integer;
begin
  if (GetDim=v.GetDim) then begin
    p:=new(PVector,Init(GetDim));
    for i:=1 to GetDim do
      p^.SetV(i,GetV(i)+v.GetV(i));
    Add:=p;
  end else
    Add:=nil;
end;

function Vector.Sub(v:vector):PVector;
var p:pvector;
    i:integer;
begin
  if (GetDim=v.GetDim) then begin
    p:=new(PVector,Init(GetDim));
    for i:=1 to GetDim do
      p^.SetV(i,GetV(i)-v.GetV(i));
    Sub:=p;
  end else
    Sub:=nil;
end;

function Vector.Scalar(v:vector):real;
var s:real;
    i:integer;
begin
  if (GetDim=v.GetDim) then begin
    s:=0;
    for i:=1 to GetDim do
      s:=s+GetV(i)*v.GetV(i);
    Scalar:=s;
  end else
    Scalar:=0;
end;

procedure Vector.Print;
var i:integer;
begin
  write('(');
```

```
  for i:=1 to GetDim do begin
    write(GetV(i));
    if (i<GetDim) then
      write(',');
  end;
  write(')');
end;

{------------------------------------------------------------}

constructor Point.Init;
begin
  Vector.Init(2);
end;

procedure Point.SetX(v:real);
begin
  SetV(1,v);
end;

procedure Point.SetY(v:real);
begin
  SetV(2,v);
end;


function Point.GetX:real;
begin
  GetX:=GetV(1);
end;

function Point.GetY:real;
begin
  GetY:=GetV(1);
end;

{------------------------------------------------------------}

var av,bv,cv:Vector;
    ap,bp,cp:Point;

begin
  av.Init(2);
  av.SetV(1,1);
```

```
  av.SetV(2,2);
  bv.Init(2);
  bv.SetV(1,3);
  bv.SetV(2,4);
  cv.Init(2);

  cv:=av.Add(bv)^;
  write('av+bv='); cv.Print; writeln;
  cv:=av.Sub(bv)^;
  write('av-bv='); cv.Print; writeln;
  writeln('av*bv=',av.Scalar(bv));

  ap.Init;
  ap.SetX(1);
  ap.SetY(2);
  bp.Init;
  bp.SetX(3);
  bp.SetY(4);
  cp.Init;

  cp:=PPoint(ap.Add(bp))^;
  write('ap+bp='); cp.Print; writeln;
  cp:=PPoint(ap.Sub(bp))^;
  write('ap-bp='); cp.Print; writeln;
  writeln('ap*bp=',ap.Scalar(bp));
end.
```

C.2 Oberon

Vector.Mod

```
(*-------------------------------------------------------------------
    Modul Vector, (C) Nils Schmeisser, 1997
    Entfernen Sie Instanzen veralteter Quellen aus dem Speicher:
    " System.Free Vector "
    Setzen Sie den star in den Rahmen und Uebersetzen Sie mit
    Compiler.Compile *
    In Ihrer Applikation importieren Sie:   IMPORT OOPVECTOR
       definieren Sie Variablen vom Typ Vector weisen Sie diesen
       Referenzen auf, mit NewVector(Dimension) erzeugte Instanzen
       zu Funktionalitaet:
       erster Parameter ist grundsaetzlich die rufende Instanz
       (Sie muessen also den Variablennamen vor dem "." immer als
       ersten Parameter ubergeben (Bsp.: c:=a-b -> c:=a.Sub(a,b);

       Vector.Get(Vector,Index)          gibt den Wert der i-ten
                                         Komponente zurueck
       Vector.Set(Vector,Index,Value)    setzt die i-te Komponente
                                         auf Value
       Vector.Add(Vector,Vector):Vector  addiert die beiden Vektoren
       Vector.Sub(Vector,Vector):Vector  subtrahiert die beiden
                                         Vektoren
       Vector.Scalar(Vector,Vector):REAL bildet das Skalarprodukt
                                         der beiden Vektoren
       NewVector(Dimension):Vector       erzeugt einen neuen Vektor
                                         (Instanz) mit der
                                         angegebenen Dimension
       Vector.Print(Vector)              drucken des Argumentes
-------------------------------------------------------------------*)

MODULE Vector;

IMPORT Out;

TYPE   Vector*=POINTER TO VectorDesc;
       VectorDesc*=RECORD
                 (*--- Attribute -------------------------------*)
                    dim:INTEGER;
                    value:POINTER TO ARRAY OF REAL;
                 (*--- die Methoden ----------------------------*)
                    Set*:PROCEDURE(a:Vector; i:INTEGER; x:REAL);
```

```
                       Get*:PROCEDURE(a:Vector; i:INTEGER):REAL;
                       Add*:PROCEDURE(a,b:Vector):Vector;
                       Sub*:PROCEDURE(a,b:Vector):Vector;
                       Scalar*:PROCEDURE(a,b:Vector):REAL;
                       Print*:PROCEDURE(a:Vector);
                   END;

(*--- forward Deklaration von NewVector ist hier noetig -----------*)

PROCEDURE ^ NewVector*(dimension:INTEGER):Vector;

(*--- Addition ----------------------------------------------------*)

PROCEDURE Add(a,b:Vector):Vector;
VAR c:Vector;
    x:REAL;
    i:INTEGER;
BEGIN
  IF (a.dim=b.dim) THEN
    c:=NewVector(a.dim);
    i:=0;
    REPEAT
      x:=a.value[i]+b.value[i];
      c.value[i]:=x;
      i:=i+1;
    UNTIL (i=a.dim);
    RETURN c;
  ELSE
    Out.String("VECTOR.Add: Dimension mismatch");
    Out.Ln;
    RETURN NIL;
  END;
END Add;

(*--- Subtraktion -------------------------------------------------*)

PROCEDURE Sub(a,b:Vector):Vector;
VAR c:Vector;
    x:REAL;
    i:INTEGER;
BEGIN
  IF (a.dim=b.dim) THEN
    c:=NewVector(a.dim);
    i:=0;
```

```
    REPEAT
      x:=a.value[i]-b.value[i];
      c.value[i]:=x;
      i:=i+1;
    UNTIL (i=a.dim);
    RETURN c;
  ELSE
    Out.String("VECTOR.Sub: Dimension mismatch");
    Out.Ln;
    RETURN NIL;
  END;
END Sub;

(*--- Skalarprodukt ----------------------------------------------*)

PROCEDURE Scalar(a,b:Vector):REAL;
VAR s:REAL;
    i:INTEGER;
BEGIN
  IF (a.dim=b.dim) THEN
    i:=0;
    s:=0;
    REPEAT
      s:=s+(a.value[i]*b.value[i]);
      i:=i+1;
    UNTIL (i=a.dim);
    RETURN s;
  ELSE
    Out.String("VECTOR.Scalar: Dimension mismatch");
    Out.Ln;
    RETURN 0;
  END;
END Scalar;

(*--- Abfrage einer Koordinate ------------------------------------*)

PROCEDURE Get(a:Vector; i:INTEGER):REAL;
BEGIN
  IF ((0<=i) & (i<a.dim)) THEN
    RETURN a.value[i];
  ELSE
    Out.String("VECTOR.Get: Index out of range"); Out.Ln;
  END;
END Get;
```

```
(*--- Setzen einer Koordinate -------------------------------------*)

PROCEDURE Set(a:Vector; i:INTEGER; x:REAL);
BEGIN
  IF ((0<=i) & (i<a.dim)) THEN
    a.value[i]:=x;
  ELSE
    Out.String("VECTOR.Set: Index out of range");
    Out.Ln;
  END;
END Set;

(*--- Drucken -----------------------------------------------------*)

PROCEDURE Print(a:Vector);
VAR i:INTEGER;
BEGIN
  Out.Char('(');
  i:=0;
  REPEAT
    Out.Real(a.value[i],10);
    IF (i<a.dim-1) THEN
      Out.Char(',');
    END;
    i:=i+1;
  UNTIL (i=a.dim);
  Out.Char(')');
END Print;

(*--- Konstruktor -------------------------------------------------*)

PROCEDURE Constructor*(dimension:INTEGER; VAR v:Vector);
VAR  i:INTEGER;
BEGIN
  v.dim:=dimension;
  NEW(v.value,v.dim);
  i:=0;
  REPEAT
    v.value[i]:=0;
    i:=i+1;
  UNTIL (i=v.dim);
  v.Set:=Set;
  v.Get:=Get;
```

```
  v.Add:=Add;
  v.Sub:=Sub;
  v.Scalar:=Scalar;
  v.Print:=Print;
END Constructor;

PROCEDURE NewVector*(dimension:INTEGER):Vector;
VAR v:Vector;
    i:INTEGER;
BEGIN
  NEW(v);
  Constructor(dimension,v);

  RETURN v;
END NewVector;

(*--- Kopieren  ---------------------------------------------------*)

PROCEDURE Copy*(a:Vector; VAR b:Vector);
VAR i:INTEGER;
BEGIN
  b.dim:=a.dim;
  NEW(b.value,b.dim);
  i:=0;
  REPEAT
    b.value[i]:=a.value[i];
    i:=i+1;
  UNTIL (i=a.dim);
  b.Set:=a.Set;
  b.Get:=a.Get;
  b.Add:=a.Add;
  b.Sub:=a.Sub;
  b.Scalar:=a.Scalar;
  b.Print:=a.Print;
END Copy;

BEGIN
END Vector.
```

Point.Mod

```
(*-----------------------------------------------------------------
    Modul OOPPOINT,  (C) Nils Schmeisser, 1997
    Entfernen Sie Instanzen veralteter Quellen aus dem Speicher:
    " System.Free Point "
    Setzen Sie den star in den Rahmen und Uebersetzen Sie mit
    Compiler.Compile *
    In Ihrer Applikation importieren Sie:  IMPORT Point
    definieren Sie Variablen vom Typ Vector weisen Sie diesen
    Referenzen auf, mit NewVector(Dimension) erzeugte Instanzen zu.
    erster Parameter ist grundsaetzlich die rufende Instanz
    (Sie muessen also den Variablennamen vor dem "." immer als
    ersten Parameter ubergeben (Bsp.:    c:=a-b   ->  c:=a.Sub(a,b);
    erbt alle Methoden von Vector

    Point.SetX(Point,REAL)          setzt den x-Wert
    Point.GetX(Point)               liefert den x-Wert
    Point.SetY(Point,REAL)          setzt den y-Wert
    Point.GetY(Point)               liefert den y-Wert
    Point.NewPoint():Point          erzeugt eine neu Instanz vom Typ
                                    Point beachten Sie die Klammern
    Point.Vector2Point(Vector):Point erzeugt aus einem Vector einen
                                    Punkt diese Funktion wird
                                    benoetigt, wenn Sie

    eine Operation mit anschliessender Zuweisung ausfuehren, etwa

         VAR a,b,c:Point;
         ...
         c:=Vector2Point(a.Add(a,b));
-------------------------------------------------------------------*)

MODULE Point;

IMPORT  Vector,Out;

TYPE Point*=POINTER TO PointDesc;
              PointDesc*=RECORD(Vector.VectorDesc)
              SetX*:PROCEDURE(p:Point; x:REAL);
              GetX*:PROCEDURE(p:Point):REAL;
              SetY*:PROCEDURE(p:Point; y:REAL);
              GetY*:PROCEDURE(p:Point):REAL;
            END;
```

```
PROCEDURE SetX(p:Point; x:REAL);
BEGIN
  p.Set(p,0,x);
END SetX;


PROCEDURE GetX(p:Point):REAL;
BEGIN
  RETURN p.Get(p,0);
END GetX;

PROCEDURE SetY(p:Point; y:REAL);
BEGIN
  p.Set(p,1,y);
END SetY;

PROCEDURE GetY(p:Point):REAL;
BEGIN
  RETURN p.Get(p,1);
END GetY;

PROCEDURE Constructor*(VAR p:Point);
VAR v:Vector.Vector;
BEGIN
  v:=p;
  Vector.Constructor(2,v);
  p.SetX:=SetX;
  p.GetX:=GetX;
  p.SetY:=SetY;
  p.GetY:=GetY;
END Constructor;

PROCEDURE NewPoint*():Point;
VAR p:Point;
BEGIN
  NEW(p);
  Constructor(p);
  RETURN p;
END NewPoint;

PROCEDURE Vector2Point*(v:Vector.Vector):Point;
VAR p:Point;
    pv:Vector.Vector;
```

```
BEGIN
  p:=NewPoint();
  pv:=p;
  Vector.Copy(v,pv);
  RETURN p;
END Vector2Point;

BEGIN
END Point.
```

MyApp.Mod

```
MODULE example3;

IMPORT Point,Out;

PROCEDURE main*;
VAR a,b,c:Point.Point;
          s:REAL;
BEGIN
  a:=Point.NewPoint();
  b:=Point.NewPoint();

  a.SetX(a,1);
  a.SetY(a,2);
  b.SetX(b,3);
  b.SetY(b,4);

  c:=Point.Vector2Point(a.Add(a,b));
  Out.String("a+b=");  c.Print(c);  Out.Ln;

  c:=Point.Vector2Point(a.Sub(a,b));
  Out.String("a-b=");  c.Print(c);  Out.Ln;

  s:=a.Scalar(a,b);
  Out.String("a*b=");  Out.Real(s,10);  Out.Ln;
END main;

BEGIN
END example3.
```

C.3 Smalltalk-80

MyVector.st

```
'From Smalltalk/X, Version:2.9.3 on 20-feb-1997 at 16:15:02'!

Object subclass:#MyVector
         instanceVariableNames:'dim values'
         classVariableNames:''
         poolDictionaries:''
         category:'Test'
!

!MyVector methodsFor:'accessing'!

dim
  ^ dim
!

at:i

  0<i ifTrue: [
    i<=dim ifTrue: [
      ^values at:i.
    ]
    ifFalse: [
      ^nil.
    ]
  ]
  ifFalse: [
    ^nil.
  ]
!

put:x at:i

  0<i ifTrue: [
    i<=dim ifTrue: [
      values at:i put:x.
    ]
  ]
! !

!MyVector methodsFor:'arithmetics'!
```

```
+ b
  |i vs vb s|

  s:=MyVector new.
  s init:dim.
  i:=Integer new.
  i:=1.
  [ i<=self dim ] whileTrue: [
    vs:=self at:i.
    vb:=b at:i.
    s put:(vs+vb) at:i.
    i:=i+1.
  ].
  ^ s
!

- b
  |i vs vb d|

  d:=MyVector new.
  d init:dim.
  i:=Integer new.
  i:=1.
  [ i<=self dim ] whileTrue: [
    vs:=self at:i.
    vb:=b at:i.
    d put:(vs-vb) at:i.
    i:=i+1.
  ].
  ^d
!

* b
  |i s vs vb|

  i:=Integer new.
  i:=1.
  s:=Float new.
  s:=0.
  [ i<=self dim ] whileTrue: [
    vs:=self at:i.
    vb:=b at:i.
    s:=s+(vs*vb).
```

```
    i:=i+1.
  ].
  ^s
!

subwith: b
  |i vs vb|

  i:=Integer new.
  i:=1.
  [ i<=self dim ] whileTrue: [
    vs:=self at:i.
    vb:=b at:i.
    self put:(vs-vb) at:i.
    i:=i+1.
  ]
! !

!MyVector methodsFor:'initialization'!

init:n
  "set dim=n and create Array"

  |i|

  dim:=Integer new.
  dim:=n.
  values:=Array new:n.
  i:=Integer new.
  i:=1.
  [ i<n ] whileTrue: [
    values at:i put:0.
    i:=i+1
  ]
! !
```

MyPoint.st

```
'From Smalltalk/X, Version:2.9.3 on 20-feb-1997 at 16:14:58'!

MyVector subclass:#MyPoint
         instanceVariableNames:''
         classVariableNames:''
         poolDictionaries:''
```

```
         category:'Test'
!

!MyPoint methodsFor:'accessing'!

y:v
  super put:v at:2
!

x:v
  super put:v at:1
!

x
  ^ super at:1
!

y
  ^ super at:2
! !

!MyPoint methodsFor:'arithmetics'!

+ b
  |p s|

  p:=MyPoint new.
  p init:2.
  s:=super + b.
  p x:(s at:1).
  p y:(s at:2).
  ^p
!

- b
  |p s|

  p:=MyPoint new.
  p init:2.
  s:=super - b.
  p x:(s at:1).
  p y:(s at:2).
  ^p
! !
```

```
!MyPoint methodsFor:'initialization'!

init
  ^ super init:2
! !
```

Example.h

```
| a b c |
a:=MyVector new.
b:=MyVector new.
a init: 2.
a put:1 at:1.
a put:2 at:2.
b init: 2.
b put:3 at:1.
b put:4 at:2.
c:=a+b.
c at:1.
c at:2.
c:=a-b.
c at:1.
c at:2.
a*b.

| a b c |
a:=MyPoint new.
b:=MyPoint new.
a init.
a x:1.
a y:2.
b init.
b x:3.
b y:4.
c:=a+b.
c x.
c y.
c:=a-b.
c x.
c y.
a*b.
```

C.4 C++

MyVector.h

```
#ifndef MyVector_
#define MyVector_

#include <ostream.h>

class MyVector {
  protected:
    unsigned int dim;
    double *value;
  public:
    MyVector(unsigned int d);
    unsigned int Dim();
    double at(unsigned int i);
    void put(double v,unsigned int i);
    MyVector &operator =(MyVector &b);
    friend MyVector &operator +(const MyVector &a,const MyVector &b);
    friend MyVector &operator -(const MyVector &a,const MyVector &b);
    friend double operator *(const MyVector &a,const MyVector &b);
    ~MyVector();

    friend ostream &operator<<(ostream &str,const MyVector &x);
};

#endif
```

MyVector.c

```
#include "MyVector.h"
#include <iostream.h>

MyVector::MyVector(unsigned int d) {
  dim=d;
  value=new double [dim];
}

unsigned int MyVector::Dim() {
  return dim;
}

double MyVector::at(unsigned int i) {
  if (i>=dim) {
```

```
    cout << "MyVector: index out of range in " << this
         << "::at(" << i << ")\n";
    exit(-1);
  }
  return value[i];
}

void MyVector::put(double v,unsigned int i) {
  if (i>=dim) {
    cout << "MyVector: index out of range in " << *this << ")::put("
         << v << "," << i << ")\n";
    exit(-1);
  }
  value[i]=v;
}

MyVector &MyVector::operator =(MyVector &b) {
  if (value!=NULL) delete value;
  dim=b.dim;
  value=new double[dim];
  for (unsigned int i=0;i<dim;i++) put(b.at(i),i);
  return *this;
}

MyVector &operator +(const MyVector &a,const MyVector &b) {
  if (a.dim!=b.dim) {
    cout << "MyVector: dimension mismatch in operator +(" << a << ","
         << b << ")\n";
    exit(-1);
  }
  MyVector *c=new MyVector(a.dim);
  for (unsigned int i=0;i<a.dim;i++) c->put(a.value[i]+b.value[i],i);
  return *c;
}

MyVector &operator -(const MyVector &a,const MyVector &b) {
  if (a.dim!=b.dim) {
    cout << "MyVector: dimension mismatch in operator -(" << a << ","
         << b << ")\n";
    exit(-1);
  }
  MyVector *c=new MyVector(a.dim);
  for (unsigned int i=0;i<a.dim;i++) c->put(a.value[i]-b.value[i],i);
  return *c;
```

```
}

double operator *(const MyVector &a,const MyVector &b) {
  if (a.dim!=b.dim) {
    cout << "MyVector: dimension mismatch in " << a << "*" << b << "\n";
    exit(-1);
  }
  double s=0.0;
  for (unsigned int i=0;i<a.dim;i++) {
    s+=a.value[i]*b.value[i];
  }
  return s;
}

MyVector::~MyVector() {
  delete value;
}

ostream &operator<<(ostream &str,const MyVector &v) {
  str << "(";
  str << v.dim << ",";
  for (int i=1;i<v.dim;i++) {
    str << v.value[i];
    if (i<v.dim-1) str << " ";
  }
  str << ")";
  return str;
}
```

MyPoint.h

```
#ifndef MyPoint_
#define MyPoint_

#include "MyVector.h"

#include <ostream.h>

class MyPoint:public MyVector {
  public:
    MyPoint();
    MyPoint(MyVector v);
    void x(double v);
    double x();
```

```
    void y(double v);
    double y();
    MyPoint &operator =(MyPoint &b);
    MyPoint &operator =(MyVector &b);
    friend MyVector &operator +(const MyPoint &a,const MyPoint &b);
    friend MyVector &operator -(const MyPoint &a,const MyPoint &b);
    friend double operator *(const MyPoint &a,const MyPoint &b);

    friend ostream &operator<<(ostream &str,const MyPoint &x);
};

#endif
```

MyPoint.c

```
#include "MyPoint.h"
#include "MyVector.h"

#include <ostream.h>

MyPoint::MyPoint():MyVector(2) {
}

MyPoint::MyPoint(MyVector v):MyVector(2) {
  value[0]=v.at(0);
  value[1]=v.at(1);
}

void MyPoint::x(double v) {
  put(v,0);
}

double MyPoint::x() {
  return at(0);
}

void MyPoint::y(double v) {
  put(v,1);
}

double MyPoint::y() {
  return at(1);
}
```

```
MyPoint &MyPoint::operator =(MyPoint &b) {
  if (value!=NULL) delete value;
  dim=b.dim;
  value=new double[dim];
  for (unsigned int i=0;i<dim;i++) put(b.at(i),i);
  return *this;
}

MyPoint &MyPoint::operator =(MyVector &b) {
  if (Dim()!=b.Dim()) {
    cout << "MyPoint: dimension mismatch in " << this << ")::=("
         << b << ")\n";
    exit(-1);
  }
  for (unsigned int i=0;i<dim;i++) put(b.at(i),i);
  return *this;
}

MyVector &operator +(const MyPoint &a,const MyPoint &b) {
  return *(new MyPoint(*(MyVector*)(&a)+*(MyVector*)(&b)));
}

MyVector &operator -(const MyPoint &a,const MyPoint &b) {
  return *(new MyPoint(*(MyVector*)(&a)-*(MyVector*)(&b)));
}

double operator *(const MyPoint &a,const MyPoint &b) {
  return (MyVector)a*(MyVector)b;
}

ostream &operator<<(ostream &str,const MyPoint &v) {
  str << (MyVector)v;
  return str;
}
```

MyApp.c

```
#include <iostream.h>
#include "MyVector.h"
#include "MyPoint.h"

main() {
  MyVector a(2),b(2),c(2);
```

```
  a.put(1,0);
  a.put(2,1);
  b.put(3,0);
  b.put(4,1);

  c=a+b;
  cout << "a+b=" << c << "\n";
  c=a-b;
  cout << "a-b=" << c << "\n";
  cout << "a*b=" << a*b << "\n";

  MyPoint ap,bp,cp;

  ap.x(1);
  ap.y(2);
  bp.x(3);
  bp.y(4);

  cp=ap+bp;
  cout << "ap+bp=" << cp << "\n";
  cp=ap-bp;
  cout << "ap-bp=" << cp << "\n";
  cout << "ap*bp=" << ap*bp << "\n";
}
```

C.5 Java

MyVector.java

```
package vector;

public class MyVector {
  int dim;
  double value[];

  public MyVector(int d) {
    dim=d;
    value=new double [dim];
  }

  public int Dim() { return dim; }

  public double at(int i) {
    if (i>=dim) {
      System.out.println("MyVector: index out of range in "+this
                         +"::at("+i+")");
    }
    return value[i];
  }

  public void put(double v,int i) {
    if (i>=dim) {
      System.out.println("MyVector: index out of range in "+this
                         +"::put("+v+","+i+")");
    }
    value[i]=v;
  }

  public MyVector plus(MyVector b) {
    if (dim!=b.dim) {
      System.out.println("MyVector: dimension mismatch in operator +("
                         +this+","+b+")");
    }
    MyVector c=new MyVector(dim);
    for (int i=0;i<dim;i++) c.put(at(i)+b.at(i),i);
    return c;
  }
```

```
  public MyVector minus(MyVector b) {
    if (dim!=b.dim) {
      System.out.println("MyVector: dimension mismatch in operator +("
                         +this+","+b+")");
    }
    MyVector c=new MyVector(dim);
    for (int i=0;i<dim;i++) c.put(at(i)-b.at(i),i);
    return c;
  }

  public double scalar(MyVector b) {
    if (dim!=b.dim) {
      System.out.println("MyVector: dimension mismatch in "
                         +this+"*"+b);
    }
    double s=0.0;
    for (int i=0;i<dim;i++) {
      s+=at(i)*b.at(i);
    }
    return s;
  }

  public String toString() {
    String s;

    s=new String();
    s=s+"(";
    for (int i=0;i<dim;i++) {
      s=s+String.valueOf(value[i]);
      if (i<dim-1)
        s=s+",";
    }
    s=s+")";
    return s;
  }
}
```

MyPoint.java

```
package vector;

public class MyPoint extends MyVector {

  public MyPoint() {
    super(2);
  }

  public void x(double v) {
    super.put(v,0);
  }

  public void y(double v) {
    super.put(v,1);
  }

  public double y() {
    return super.at(1);
  }

}
```

MyApp.java

```
import java.applet.*;

import vector.*;

public class MyApp extends Applet {

  public static void main(String args[]) {
    MyVector a,b,c;
    MyPoint ap,bp,cp;

    a=new MyVector(2);
    a.put(1,0);
    a.put(2,1);
    b=new MyVector(2);
    b.put(3,0);
    b.put(4,1);
    System.out.println("a+b="+a.plus(b));
    System.out.println("a-b="+a.minus(b));
    System.out.println("a*b="+a.scalar(b));
```

```
    ap=new MyPoint();
    ap.x(1);
    ap.y(2);
    bp=new MyPoint();
    bp.x(3);
    bp.y(4);
    System.out.println("ap+bp="+ap.plus(bp));
    System.out.println("ap-bp="+ap.minus(bp));
    System.out.println("ap*bp="+ap.scalar(bp));
  }
}
```

C.6 Ada 95

linalg.ads

```
package linalg is

  type vecpack is private;

  subtype vector is vecpack;

  function NewVector(d:integer) return vector;
  function dimension(v:vector) return integer;
  procedure put(v:out vector; i:integer; x:float);
  function get(v:vector; i:integer) return float;

  function "+" (a,b:vector) return vector;
  function "-" (a,b:vector) return vector;
  function "*" (a,b:vector) return float;

  procedure print(v:vector);

  private

    type vecarray is array (integer range <>) of float;

    type vecpack is
    record
      dim:integer;
      value:vecarray(1..100);
    end record;

end linalg;
```

linalg.adb

```
with Text_IO;
use Text_IO;

package body linalg is

  function dimension(v:vector) return integer is
  begin
    return v.dim;
  end dimension;
```

```
procedure create(v:out vector; d:integer) is
begin
  v.dim:=d;
end create;

function NewVector(d:integer) return vector is
  v:vector;
begin
  create(v,d);
  return v;
end NewVector;

function get(v:vector; i:integer) return float is
begin
  return v.value(i);
end get;

procedure put(v:out vector; i:integer; x:float) is
begin
  v.value(i):=x;
end put;

function "+" (a,b:vector) return vector is
  c:vector;
  d,i:integer;
begin
  d:=dimension(a);
  create(c,d);
  i:=1;
  for i in 1..d loop
    put(c,i,get(a,i)+get(b,i));
  end loop;
  return c;
end;

function "-" (a,b:vector) return vector is
  c:vector;
  d,i:integer;
begin
  d:=dimension(a);
  create(c,d);
  i:=1;
  for i in 1..d loop
    put(c,i,get(a,i)-get(b,i));
```

```
    end loop;
    return c;
  end;

  function "*" (a,b:vector) return float is
    d,i:integer;
    s:float;
  begin
    d:=dimension(a);
    i:=1;
    s:=0.0;
    for i in 1..d loop
      s:=s+get(a,i)*get(b,i);
    end loop;
    return s;
  end;

  procedure print(v:vector) is
    i,d:integer;
    package Flt_Io is new Text_IO.Float_IO (Float);
    use Flt_Io;
  begin
    Text_IO.put("(");
    d:=dimension(v);
    i:=1;
    for i in 1..d loop
      put(get(v,i));
      if (i<d) then
        Text_IO.put(",");
      end if;
    end loop;
    Text_IO.put(")");
  end print;

end linalg;
```

geometry.ads

```
with linalg;
use linalg;

package geometry is

  subtype point is vector;
```

```
  function NewPoint return point;

  function GetX(p:point) return float;
  procedure SetX(p:out point; x:float);
  function GetY(p:point) return float;
  procedure SetY(p:out point; x:float);

end geometry;
```

geometry.adb

```
with linalg;
use linalg;

package body geometry is

  function NewPoint return point is
    p:point;
  begin
    p:=NewVector(2);
    return p;
  end NewPoint;

  function GetX(p:point) return float is
  begin
    return get(p,1);
  end GetX;

  procedure SetX(p:out point; x:float) is
  begin
    put(p,1,x);
  end SetX;

  function GetY(p:point) return float is
  begin
    return get(p,2);
  end GetY;

  procedure SetY(p:out point; x:float) is
  begin
    put(p,2,x);
  end SetY;

end geometry;
```

main.adb

```
with Text_IO,linalg,geometry;
use Text_IO,linalg,geometry;

procedure main is
  a,b,c:vector;
  ap,bp,cp:vector;
  s:float;
  package Flt_Io is new Float_IO (Float);
  use Flt_Io;
begin
  a:=NewVector(2);
  b:=NewVector(2);
  put(a,1,1.0);
  put(a,2,2.0);
  put(b,1,3.0);
  put(b,2,4.0);

  c:=a+b;
  put("a+b="); print(c); new_line;
  c:=a-b;
  put("a-b="); print(c); new_line;

  s:=a*b; put("a*b="); put(s); new_line;

  ap:=NewVector(2);
  bp:=NewVector(2);
  SetX(ap,1.0);
  SetY(ap,2.0);
  SetX(bp,3.0);
  SetY(bp,4.0);

  cp:=ap+bp;
  put("ap+bp="); print(cp); new_line;
  cp:=ap-bp;
  put("ap-bp="); print(cp); new_line;

  s:=ap*bp; put("ap*bp="); put(s); new_line;
end main;
```

C.7 Eiffel

MyVector.e

```
class MyVector

creation
   Init

feature
   io : BASIC_IO
   dimension:integer
   value:array[integer]

   Init(i:INTEGER) is
      do
   dimension:=i
   !!value.make(1,i)
   !!io
      end

   Dim:integer is
      do
   result:=dimension
      end

   put(x:real,i:integer) is
      do
   value.put(x,i)
      end

   at(i:integer):real is
      do
   result:=value.item(i);
      end

   infix "+",Plus (b:MyVector):MyVector is
      require
   b.Dim=Dim
      local i:integer
   v:MyVector
      do
   !!v.Init(dimension)
   from
```

```
   i:=1
until
   i>dimension
loop
   v.Put(at(i)+b.at(i),i)
   i:=i+1
end
result:=v
   end

infix "-",Minus (b:MyVector):MyVector is
   require
b.Dim=Dim
   local i:integer
v:MyVector
   do
!!v.Init(dimension)
from
   i:=1
until
   i>dimension
loop
   v.Put(at(i)-b.at(i),i)
   i:=i+1
end
result:=v
   end

infix "*",scalar (b:MyVector):real is
   require
b.Dim=Dim
   local i:integer
s:real
   do
s:=0
from
   i:=1
until
   i>dimension
loop
   s:=s+at(i)*b.at(i)
   i:=i+1
end
result:=s
```

```
      end

   print is
      local
   i:integer
      do
   io.put_char ('(')
   from
      i:=1
   until
      i>dimension
   loop
      io.put_real(at(i))
            if i<dimension then
         io.put_char(',')
      end
      i:=i+1
   end
   io.put_char (')')
      end

end
```

MyPoint.e

```
class MyPoint

inherit MyVector
      rename
   Init as VectorInit,
   Plus as VectorPlus,
   Minus as VectorMinus
      redefine
   infix "+",infix "-"
      end

creation
   Init

feature

   Init is
      do
   VectorInit(2)
      end
```

```
   SetX(v:real) is
      do
   put(v,1)
      end

   GetX:real is
      do
   result:=at(1)
      end

   SetY(v:real) is
      do
   put(v,2)
      end

   GetY:real is
      do
   result:=at(2)
      end

   infix "+",Plus (b:MyPoint):MyPoint is
      local
   c:MyVector
      do
   !!c.init(2)
   c:=current.VectorPlus(b)
   !!result.init
   result.put(c.at(1),1)
   result.put(c.at(2),2)
      end

   infix "-",Minus (b:MyPoint):MyPoint is
      local
   c:MyVector
      do
   !!c.init(2)
   c:=current.VectorMinus(b)
   !!result.init
   result.put(c.at(1),1)
   result.put(c.at(2),2)
      end

end
```

MyApp.e

```
class MyApp

creation
   make

feature {ANY}

   io:BASIC_IO

   make is
      local
   a,b,c:MyVector
   ap,bp,cp:MyPoint
      do
   !!io
   !!a.Init(2)
   !!b.Init(2)
   !!c.Init(2)

   a.put(1,1)
   a.put(2,2)
   b.put(3,1)
   b.put(4,2)

   io.put_string("a+b=")
   c:=a+b
   c.print
   io.put_string("%N")

   io.put_string("a-b=")
   c:=a-b
   c.print
   io.put_string("%N")

   io.put_string("a*b=")
   io.put_real(a*b)
   io.put_string("%N")

   !!ap.Init
   !!bp.Init
   !!cp.Init
```

```
   ap.SetX(1)
   ap.SetY(2)
   bp.SetX(3)
   bp.SetY(4)

   io.put_string("ap+bp=")
   cp:=ap+bp
   cp.print
   io.put_string("%N")

   io.put_string("ap-bp=")
   cp:=ap-bp
   cp.print
   io.put_string("%N")

   io.put_string("ap*bp=")
   io.put_real(ap*bp)
   io.put_string("%N")

      end
end
```

MyApp.pdl

```
program MyApp

root
    MyApp : "make"

cluster

    "./"
    end

include "$EIFFEL_S/library/lib.pdl"
include "$EIFFEL_S/library/lib.lib"

end
```

Anhang D

Autoren

Sören Auer Stud.-Math., e-mail: auer@rcs.urz.tu-dresden.de

M. Zouhir Barakat Stud.-Math.

Anja Fröhner Stud.-Math., e-mail: froehner@math.tu-dresden.de

Ismail Idriss Dipl.-Math., e-mail: idriss@rcs.urz.tu-dresden.de

Dominik Kostanjšek Stud.-Math.

Andreas Knüpfer Stud.-Math., e-mail: knuepfer@atlantis.wh2.tu-dresden.de

Jürgen Lampe Doz. Dr. rer. nat., e-mail: lampe@math.tu-dresden.de

Ingmar Meinecke Stud.-Math., e-mail: im1@rcs.urz.tu-dresden.de

Andreas Schälicke Stud.-Phys., e-mail: andreas@x14l1b.wh6.tu-dresden.de

Tino Seifert

Nils Schmeißer Dipl.-Math., e-mail: nils@schmeisser.com

Wolfgang V. Walter Prof. Dr. rer. nat., e-mail: wwalter@math.tu-dresden.de

Anhang E

Bild- und Literaturnachweis

Abbildungsverzeichnis

1.1 Stammbaum von Programmiersprachen 6

2.1 Ableitungsschema nach [Mey89] 9
2.2 "Naives Speichermodell" nach [AC96] 14
2.3 Methodenaufruf und Methodensammlung 14
2.4 Schematische Darstellung des flachen Kopierens 21
2.5 Prinzip der dynamischen Vererbung 22
2.6 Klassenbildung durch Prototypen und Traits 22

3.1 Klassendefinition 25
3.2 statische und dynamische Instanzen und Zugriff 27
3.3 Der Collection-Teilbaum in Smalltalk 45
3.4 Smalltalk-80 Systemarchitektur 49

4.1 Gesichtspunkte beim Design der "Alto" Oberfläche (nach [Heu92]) 88
4.2 Der Aufbau von Windows NT (aus [Rod93]) 89
4.3 ORB Konzept 90
4.4 Relation Personen - Person "wohnt in" Ort 93
4.5 Relation Hobbys - Person hat Hobby 93
4.6 generische Operationen für OODMs 98

A.1 Borland DOS-IDE 101
A.2 Oberon Oberfläche 102
A.3 Belegungen der Maustasten unter Oberon, die Zeile gibt die zuerst gedrückte Taste an 103
A.4 Smalltalk Oberfläche 104

Literaturverzeichnis

[1] Bjarne Stroustrup:
"The C++ Programming Language";
Addison-Wesley; 1993

[2] Internatial Business Machine:
"IBM C Set++ for AIX: Language Reference; Version 3 Release 1";
IBM; 1994

[3] Borland:
"Borland C++: Programming Guide";
Borland International

[4] Fritz Jobst:
"Programmieren in Java";
Carl Hanser Verlag; 1996

[5] Ken Arnold, James Gosling:
"Java^{TM} - Die Programmiersprache";
Eddison-Wesley; 1996

[6] Sun Microsystems:
"The Java Virtual Machine Specification, Release 1.0 Beta, Draft";
Sun Microsystems Computer Corp.; August 21st, 1995

[7] Middendorf, Singer, Strobel:
"JAVA Programmier- und Referenzhandbuch";
Verlag für digitale Technologie GmbH

[8] "The Java Tutorial";
http://java.sun.com/nav/read/Tutorial/

[9] Davignon und Edelmann:
"Java, oder wie steuere ich meine Kaffeemaschine?";
tewi Verlag GmbH

[10] PC Professionell 2/97;
ZIFF DAVIS Verlag; 1997

[11] Nils Schmeißer:
"Nebenläufigkeit in Java formuliert, Thread-Konzept, Synchronisationsmechanismen";
interner Report, FVTK, FZR; 1997

[12] Nils Schmeißer:
"Forschungszentrum Rossendorf - Standortinformation im WWW";
interner Report, FVTK, FZR; 1997

[13] Hans-Jürgen Hoffmann:
"Smalltalk verstehen und anwenden";
Carl Hanser Verlag; 1987

[14] Adele Goldberg, David Robson: "Smalltalk-80 – the language";
Addison-Wesley Publishing Company; 1989

[15] "Smalltalk/V286 – Tutorial and Programming Handbook";
Digitalk Inc.; 1988

[16] Intuitive Systems Limited:
"Dolphin-Smalltalk development system";
http://www.intuitive.co.uk/dolphin/dolphin.htm

[17] Dean W. Gonzalez:
"Ada Programmer's Handbook and Language Reference Manual";
The Benjamin/Cummings Publishing Company; 1991

[18] J.G.P. Barnes:
"Programmieren in Ada";
Carl Hanser Verlag; 1982

[19] C. Comar, F. Gasperoni, E. Schonberg:
"The GNAT Project: A GNU-Ada9X Compiler";
Courant Institute of Mathematical Science, New York University

[20] K. Jensen, N. Wirth:
"PASCAL - User Manual and Report";
Springer-Verlag; 1985

[21] A. Schäpers u.a.:
"Borland Pascal 7.0 mit Objekten, Programmierhandbuch";
Borland GmbH; 1993

[22] Martin Reiser, Nikolaus Wirth:
"Programming in Oberon, Steps beyond Pascal and Modula";
Addison-Wesley; 1992

[23] Martin Reiser, Niklaus Wirth:
"Programmieren in Oberon. Das neue Pascal.";
Addison-Wesley; 1994

[24] Moessenboeck:
"Objektorientierte Programmierung in Oberon-2";
Springer-Verlag, 1994

[25] Oberon-HomePage: http://ics.inf.ethz.ch/Oberon.html

[26] Oberon-microsystems-HomePage: http://www.oberon.ch/

[27] Robert Switzer:
"Eiffel: An Introduction";
Prentice Hall, 1993

[28] Richard Wiener:
"An Object-Oriented Introduction to Computer Science Using Eiffel";
Prentice Hall, 1996

[29] "Eiffel: An overview of the language and method";
http://www.eiffel.com/doc/manuals/language/intro/BOOK.maker.html

[30] Betrand Meyer:
"Eiffel: The Language";
Prentice Hall; 1992

[31] Wilhelm Gehrke:
"Fortran95 Language Guide";
ISBN 3-540-76062-8; 1996

[32] S. Ramsden, F. Lin, M.A. Pettipher, G.S. Noland, J.M. Brooke:
"Fortran 90" (Student Notes);
via World Wide Web; 1995

[33] RRZN Hannover:
"Fortran 90";
DIN EN 21539-1992; 1993

[34] Wolfgang V. Walter:
"The Lectures of Informatic (1 & 2) for academic year 1996/1997";
Lecture Service of the Institute of Scientific Computing, TU Dresden; 1997

[35] Miles Ellis:
"The final content of Fortran 2000 is determined !";
via World Wide Web from University of Oxford; 1997

[36] Ali Jamal Al-Dien:
"Fortran Language";
University of Damascus; 1994

[37] Andreas Heuer:
"Objektorientierte Datenbanken";
Addison-Wesley; 1992

[38] Thomas Blummer:
"Objektverwalter (Objektdatenbanken - High-Tech-Spielzeug oder Zukunftsmodell)";
c't Mai 97; Heise Verlag; 1997

[39] "ODMG";
`http://www.odmg.org`

[40] Dr. Gerd Wagner:
"Jenseits von Schema F (Datenmodelle - Strickmuster für Datenbanken)";
c't Mai 97; Heise Verlag; 1997

[41] Sebesta:
"Concepts of Programming Languages.";
Benjamin/Cummings Publ Co, 1993

[42] Carl A. Gunter, John C. Mitchell:
"Theoretical Apsects of Object-Oriented Programming";
The MIT Press, 1994

[43] Bal, Grune:
"Programming Languages Essentials";
Eddison-Wesley, 1994

[44] Wolff von Gudenberg:
"Objektorientiert Programmieren von Anfang an.";
BI-Wissenschaftsverlag, 1993

[45] Bertand Meyer:
"Object-Oriented Software Construction";
Prentice Hall, 1989

[46] Kurt-Ullrich Witt:
"Einführung in die objektorientierte Programmierung";
R. Oldenburg Verlag, 1992

[47] Horst Reichel:
"An approach to object semantics based on terminal co-algebras";
Mathematical Structures in Computer Science, 5:129-152; 1995

[48] Ulrich Hensel:
"Statische und dynamische Eigenschaften von Elementen finaler Coalgebren";
Diplomarbeit in der Fakultät Informatik der TU Dresden; 1994

[49] J. Bergin, M. Stehlik, J. Roberts, R. Pattis:
"Karel++ - A gentle Introduction to the Art of Object-Oriented Programming";
John Wiley and Sons, 1997

[50] H.J. Rode:
"Windows NT - Von 3.1 nach NT";
Addison-Wesley; 1993

[51] Manfred Schneider:
"Some Links: Object-Orientation";
http://www.rhein-neckar.de/~cetus/software.html; 1996-1997

[52] B. Janssen, M. Spreitzer:
"ILU Reference Manual";
http://dsg.harvard.edu/public/software/ILU/manual_toc.html

[53] "CORBA and OMG Information Resources";
http://www.acl.lanl.gov/CORBA/

[54] Jens-Peter Redlich:
"CORBA 2.0 - Esperanto für Objekte?";
unix/mail 5/96 S. 319 ff.; Carl Hanser Verlag; 1996

Anhang F

Bild- und Literaturnachweis

Abbildungsverzeichnis

Literaturverzeichnis

[AC96] M. Abadi and L. Cardelli. *A Theory of Objects*. Springer-Verlag New York, Inc., 1996. object-oriented features, untyped first-order calculi, second-order calculi, higher- order calculi.

[AG96] K. Arnold and J. Gosling. *Java^{TM} - Die Programmiersprache*. Eddison-Wesley, 1996.

[Bar82] J.G.P. Barnes. *Programmieren in Ada*. Carl Hanser Verlag, 1982.

[CGS] C. Comar, F. Gasperoni, and E. Schonberg. The gnat project: A gnu-ada9x compiler. Courant Institute of Mathematical Science, New York University.

[Eif] Eiffel: An overview of the language and method.

[GM94] C.A. Gunter and J.C. Mitchell. *Theoretical Apsects of Object-Oriented Programming*. The MIT Press, 1994. sorts, (many) sorted-algebras, theory.

[Gon91] D.W. Gonzalez. *Ada Programmer's Handbook and Language Reference Manual*. The Benjamin/Cummings Publishing Company, 1991.

[Heu92] A. Heuer. *Objektorientierte Datenbanken*. Addison-Wesley, 1992.

[Hof87] H.-J. Hoffmann. *Smalltalk verstehen und anwenden*. Carl Hanser Verlag, 1987.

[JW85] K. Jensen and N. Wirth. *PASCAL - User Manual and Report*. Springer-Verlag, 1985.

[Mey89] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, 1989.

[Mey92] B. Meyer. *Eiffel: The Language*. Prentice Hall, 1992.

[Obe] Oberon-homepage.

[Red96] J.-P. Redlich. Corba 2.0 - esperanto für objekte? *unix/mail*, 5:319, 1996.

[Rod93] H.J. Rode. *Windows NT - Von 3.1 nach NT.* Addison-Wesley, 1993.

[RW92] M. Reiser and N. Wirth. *Programming in Oberon, Steps beyond Pascal and Modula.* Addison-Wesley, 1992.

[RW94] M. Reiser and N. Wirth. *Programmieren in Oberon. Das neue Pascal.* Addison-Wesley, 1994.

[Str93] B. Stroustrup. *The C++ Programming Language.* Addison-Wesley, 1993.

[Su93] A. Schäpers and u.a. *Borland Pascal 7.0 mit Objekten, Programmierhandbuch.* Borland GmbH, 1993.

[Sun95] Sun Microsystems Computer Corp. *The Java Virtual Machine Specification, Release 1.0 Beta, Draft*, 1995.

[Swi93] R. Switzer. *Eiffel: An Introduction.* Prentice Hall, 1993.

[Wie96] R. Wiener. *An Object-Oriented Introduction to Computer Science Using Eiffel.* Prentice Hall, 1996.