I. Meine Empfehlung

Ich rate dringend davon ab, die in der Spezifikationsschrift im Zusammenhang mit dem Konzept der eingebetteten Objekte vorgeschlagenen Anweisungen

`GET REFERENCE` und `ASSIGN OBJECT`

einzuführen. Denn ich sehe in dem Wunsch, die eingebetteten Objekte wie Werte behandeln zu können, einen Versuch, Unvereinbares zu vereinbaren. In den folgenden Betrachtungen werde ich dies näher ausführen.
Ebenso dringend empfehle ich **die Aufnahme des Konzepts des abstrakten Datentyps** für die Gedächtnisvariablen der Objekte.

II. Meine Begründung

1. Mögliche Gründe für die Einführung eingebetteter Objekte

An der Stelle, wo in der Spezifikationsschrift (Version 0.9.1) eingebettete Objekte eingeführt werden, findet man als Begründung folgende Hinweise:

> Man will nicht gezwungen sein, in jedem Fall alle Objekte dynamisch anlegen zu müssen.
> Man will vielmehr bestimmte Objekte, die eindeutig zu einem anderen Objekt gehören, wie Werte, also wie Strukturen mit Methoden behandeln können.

Meines Erachtens sollten hier einige mögliche Ziele auseinandergehalten werden, die man mit der Einbettung verbinden kann:

- Ein mögliches Ziel ist die Existenzbindung des eingebetteten Objekts an sein enthaltendes Objekt. Man will, daß bei der Kreation des enthaltenden Objekts das eingebettete Objekt automatisch mitgeschaffen wird, ohne daß der Programmierer hierfür in der Source explizit eine Create–Anweisung vorsehen muß.

- Ein anderes mögliches Ziel ist die Kapselung. Man will, daß das enthaltende Objekt als Kapsel für das eingebettete Objekt wirkt, so daß anderen Objekten der direkte Zugang zu dem eingebetteten Objekt verwehrt ist.

- Ein weiteres mögliches Ziel ist die Anhebung der Methodensprache, die erreicht wird, indem man im Objektgedächtnis Variable zuläßt, deren Typ als abstrakter Datentyp eingeführt wird.

Hinter allen diesen Zielen steht der Wunsch nach angemessener Strukturierung des Sourcecodes. Ich befürworte deshalb die Bemühungen um eine optimale Einführung des Einbettungsprinzips.

Warum nicht „shared variables" ??

z.B.

abstrakter (selbstdef.) Datentyp ??

2. Der Kern des Begriffs "Objekt" bei der Objektorientierung

Es ist auffällig, daß in der Literatur und in Vorträgen, wo der Objektbegriff vorgestellt wird, so gut wie immer der eigentliche Kern unausgesprochen bleibt: Man kann ein Objekt nicht als Einzelwesen definieren, denn ein einzelnes Objekt ist sinnlos. Der Objektbegriff muß über die Kommunikationsfähigkeit definiert werden:

- Ein Objekt im Sinne der Objektorientierung ist eine in ihrem Verhalten durch Software definierte Funktionseinheit, die ein Gedächtnis besitzt und die mit anderen Objekten, von deren Existenz sie weiß, über ein Kommunikationssystem kommunizieren kann.

- Die Menge der existierenden Objekte in einem laufenden System kann zeitabhängig sein, d.h. es können Objekte entstehen und verschwinden. Es muß zum Zeitpunkt des Systemstarts mindestens ein Objekt geben, welches die Fähigkeit besitzt, neue Objekte zu schaffen. In manchen Systemen haben alle Objekte diese Fähigkeit, in anderen Systemen ist diese Fähigkeit auf Objekte bestimmten Typs beschränkt.

- Unterschiedliche Objektmodelle ergeben sich aus der Möglichkeit, die Fähigkeit der Objektkreation unterschiedlich zu verteilen, und aus der Möglichkeit, das Protokoll der Objektkommunikation unterschiedlich zu gestalten..
  Allen Protokollen gemeinsam ist die Möglichkeit des Methodenaufrufs und die Möglichkeit der Lieferung eines Returnwerts. Häufig findet man auch die Möglichkeit der Kommunikation über Ereignisse, die veröffentlicht werden und die von denjenigen Objekten wahrgenommen werden, die sich zuvor darauf subskribiert haben.

Bild 1 veranschaulicht die eingeführten Elemente. In diesem Bild sieht man sieben Objekte, die über einen Zusteller – eine Art Briefträger – für Aufrufe und Returnwerte miteinander verbunden sind. In das Innere des Objekts 1 kann man hineinsehen. Für dieses Objekt sind die sechs Methoden M1 bis M6 definiert, d.h. dieses Objekt kann sechs unterschiedliche Arten von Aufträgen erledigen. Das Gedächtnis des Objekts ist in die vier Variablen V1 bis V4 gegliedert. In V1 ist das Wissen um die Existenz des Objekts 2 gespeichert.

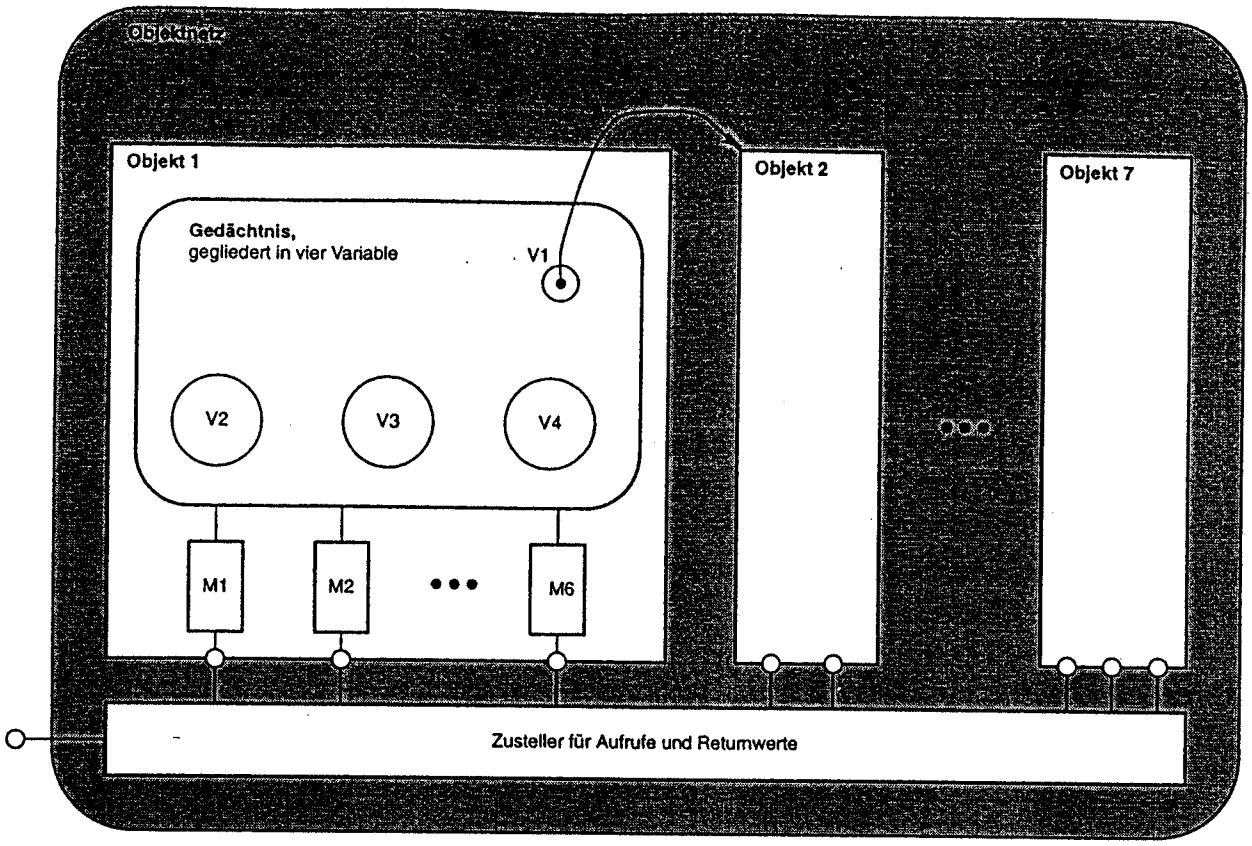
Es wurden hier bewußt die Bezeichnungen "Attribute" oder "Properties" für die Variablen im Gedächtnis vermieden, weil es in Texten über OO keine einheitliche Verwendung dieser Begriffe gibt. Häufig werden zwar die Bezeichnungen Attribut oder Property für die Gedächtnisvariablen verwendet, aber manchmal werden auch die Returnwerte von Methoden, die der Abfrage von Gedächtnisinhalten dienen, als Attribute oder Properties bezeichnet. In den Fällen, wo diese Returnwerte von Abfragemethoden nicht mit den im Gedächtnis gespeicherten Werten identisch sind, sondern aus diesen berechnet werden, ergibt sich leicht eine Verwirrung bei der Verwendung der Begriffe Attribut oder Property.

3. Konventionelle Typen für Gedächtnisvariable

In Bild 1 wurde symbolisch zum Ausdruck gebracht, daß man die Gedächtnisvariablen in zwei Klassen einteilen kann, nämlich in die Klasse der Referenzvariablen und in die Klasse der Nichtreferenzvariablen.

In den Nichtreferenzvariablen befindet sich "das eigenständige Wissen" des Objekts, welches in den Methoden des Objekts verarbeitet werden kann, ohne daß dabei eine Kommunikation mit anderen Objekten stattfinden muß. Man denke an Integers, Reals, Booleans, Strings oder Records.

Demgegenüber kann die in den Referenzvariablen enthaltene Information nicht verarbeitet, sondern nur verglichen oder in eine andere Referenzvariable kopiert oder anläßlich der Kommunikation mit einem anderen Objekt dem Zusteller mitgeteilt werden. Bei der Übergabe von Referenzinformation an den Zusteller wird diese Information entweder als Returnwert oder als Zielangabe für einen Aufruf oder als Parameter in einem Aufruf verwendet.



Interpretation der Symbole:

Akteur,
der mit den Gedächtnisinhalten, die in seinem Zugriffsbereich liegen, arbeiten kann, und der als Kommunikationspartner anderer Akteure auftreten kann

Kommunikationskanal

Gedächtnisvariable für eine Objektreferenz

Gedächtnisvariable für sonstige Daten

Bild 1 Zweierlei Arten von Variablen im Gedächtnis von Objekten

Der Unterschied zwischen Nichtreferenzvariablen und Referenzvaraiblen kann salopp durch den Spruch verdeutlicht werden: "Der eine weiß etwas, während der andere nur einen kennt, der etwas weiß."

Die Einführung von Nichtreferenzvariablen als mögliche Gedächtniszellen von Objekten ist eine Erweiterung gegenüber Smalltalk, wo das Objektgedächtnis grundsätzlich nur Referenzvariablen enthalten kann.

4. Abstrakte Typen für Gedächtnisvariable

4.1 Der Begriff des abstrakten Datentyps

Für die durch die jeweilige Programmiersprache vorgegebenen Basistypen – Integer, Real, etc. – und die aus diesen Basistypen zusammensetzbaren Strukturtypen werden durch die Programmiersprache auch die funktionalen Verknüpfungsmöglichkeiten festgelegt.

Durch die Einführung der sog. abstrakten Datentypen wurde dem Programmierer die Möglichkeit gegeben, selbst die funktionalen Verknüpfungsmöglichkeiten der von ihm definierten Typen festzulegen.

Es ist ganz wichtig zu beachten, daß in der Spezifikation abstrakter Datentypen keine Anweisungen oder Methoden, sondern Funktionen spezifiziert werden. Eine Funktion ist eine Abbildung einer Argumentmenge auf eine Ergebnismenge, wobei diese Mengen jeweils durch bestimmte Typen spezifiziert sind.

Als Beispiel eines abstrakten Datentyps wird das Lehrbuchbeispiel STACK betrachtet.

```
DEFINE ABSTRACT_DATA_TYPES
  STACK[ REPERTOIRE ]

FUNCTIONS
  STACK       clear();

  INTEGER     height( s:STACK );

  STACK       push( s:STACK, x:REPERTOIRE );

  STACK       pop( s:STACK );
                RESTRICTIONS
                  height(s)≠0;

  REPERTOIRE  top( s:STACK );
                RESTRICTIONS
                  height(s)≠0;

AXIOMS
  height( clear() ) = 0;

  height( push(s, x) ) = 1+ height(s);

  top( push(s, x) ) = x;

  pop( push(s, x) ) = s;
```

4.2 Unterscheidung zwischen Funktionen und Anweisungen

Da in einer Funktion i.a. unterschiedliche Typen miteinander verknüpft werden, macht es keinen Sinn, eine Funktion einem bestimmten Typ zuzuordnen. Man betrachte als Beispiel die folgenden beiden Funktionen:

Beispiele:

```
(1)     INTEGER   height( s:STACK )

(2)     LIST      add_item( liste:LIST, x:INTEGER )
```

Man frage sich nun einmal, mit welchen Argumenten man seine intuitive Vorstellung rechtfertigen würde, daß die Funktion `height` zum Typ `STACK` und die Funktion `add_item` zum Typ `LIST` gehören sollten.

Anders liegen die Verhältnisse im Falle von Anweisungen. Es interessieren hier nur diejenigen Anweisungen, in denen die Berechnung einer Funktion verlangt wird. In einer solchen Anweisung müssen die Angabe der zu berechnenden Funktion, die Festlegung der Argumentbelegung und die Angabe des Zielorts, an dem das Ergebnis abgeliefert werden soll, zusammengepackt sein.

Beispiele:

```
(1)     h := height(s)

(2)     return( first_item(liste) )

(3)     add_item( liste, x )
```

In den Anweisungen (1) und (2) sind alle beteiligten Operanden, d.h. sowohl alle Quellen für die Argumente als auch die Senke für das Ergebnis explizit genannt, wogegen in der Anweisung (3) nur zwei Operanden explizit aufgeführt sind, obwohl hier drei Operanden beteiligt sind, nämlich zwei Argumente und ein Ergebnis. Die Anweisung (3) kann man nur verstehen, wenn man die Vereinbarung kennt, daß das erste Funktionsargument, also der ursprüngliche Wert in der Variable `liste`, durch das Ergebnis der Funktionsberechnung überschrieben werden soll. Man hätte ja die Anweisung ohne Änderung ihrer Semantik auch in einer Form schreiben können, in der alle drei Operanden explizit genannt werden:

```
liste := add_item( liste, x )
```

In der Welt der objektorientierten Programmierung kommen zweierlei Arten von Anweisungen vor: Zum einen gibt es die Anweisungen, die ein Objekt einem anderen Objekt erteilt; diese seien *object_to_object–Anweisungen* genannt. Zum anderen gibt es die Anweisungen, die bei der Abwicklung einer Methode innerhalb eines Objekts ausgeführt werden müssen; diese seien *method_to_processor–Anweisungen* genannt.

Die Form der object_to_object–Anweisungen hängt nicht davon ab, welche Variablentypen im Gedächtnis der Objekte vorkommen. Diese Anweisungen werden vielmehr durch die Spezifikation der Methoden in der Klasse des gerufenen Objekts festgelegt.

Die Methoden, die durch die object_to_object–Anweisungen aufgerufen werden, sind also jeweils eindeutig einer Klasse zugeordnet. Demgegenüber lassen sich weder die method_to_processor–Anweisungen noch die Funktionen, deren Berechnung in diesen Anweisungen verlangt wird, mit akzeptabler Begründung eindeutig einem Datentyp zuordnen.

Zu dem Paar Methode/Klasse gibt es also kein analoges Paar Funktion/Typ oder Anweisung/Typ.

Da durch die Spezifikation von abstrakten Datentypen die Form der method_to_processor–Anweisungen, in denen diese Typen vorkommen, nicht festgelegt werden, muß die Form dieser Anweisungen noch separat festgelegt werden.

Es ist ja keinesfalls selbstverständlich, daß man in jedem Falle die zu den abstrakten Datentypen spezifizierten Funktionen in all denjenigen Formen in Anweisungen zuläßt, die für die Funktionen mit konventionellen Datentypen üblich sind. Man betrachte die folgenden Vorkommensformen für Funktionen in Anweisungen einmal unter der Annahme, anstelle der Funktion `f(...)` stehe die Funktion `height(s)`, und ein andermal unter der Annahme, anstelle von `f(...)` stehe `add_item(liste, x)`.

```
(1)     y := f(...)

(2)     IF  f(...) EQUAL y  THEN  ...

(3)     y := g(..., f(...), ...)

(4)     return( f(...) )
```

4.3 Unterscheidung zwischen Typen und Klassen

Der Gedanke liegt nahe, die Form der method_to_processor–Anweisungen an der Vorstellung zu orientieren, jede Gedächtnisvariable, deren Typ ein abstrakter Datentyp ist, bilde jeweils das Gedächtnis eines Objekts. Denn dann kann man die method_to_processor–Anweisungen wie object_to_object–Anweisungen formulieren. Allerdings braucht man dazu Referenzen auf diese "Embedded Objects". Dies birgt die Gefahr in sich, daß der Unterschied zwischen den *embedded objects* und den *peer objects* verwischt wird, d.h. daß die Kapselung des Gedächtnisinhalts eines Objekts durchbrochen wird. Denn wenn man die Referenzen auf die embedded objects genau so behandeln kann wie die Referenzen auf die peer objects, dann kann ein Objekt seinen Peers die Referenzen auf seine embedded objects mitteilen. Damit wären wesentliche Vorteile der Objektorientierung verloren.

Ich sehe keinerlei Vorteil einer solchen Entscheidung!

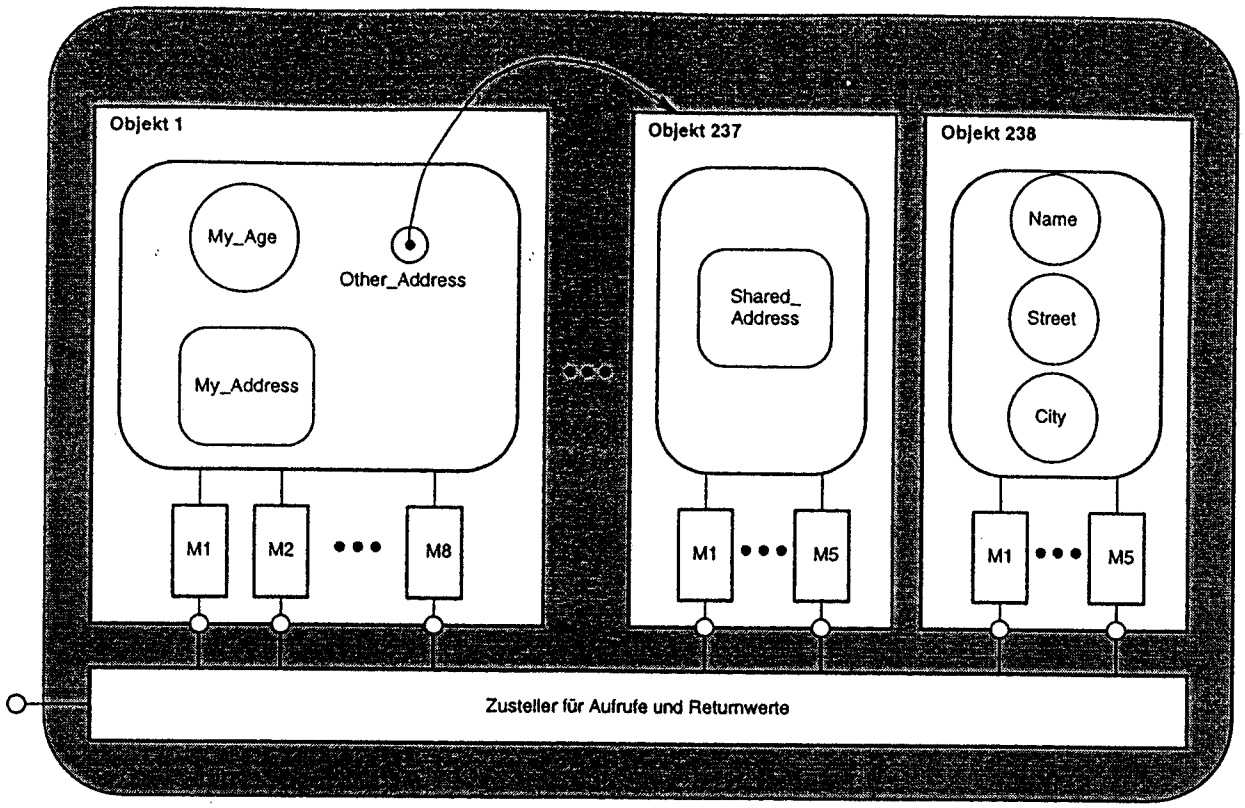
In Bild 2 ist ein Beispiel dargestellt, anhand dessen die Problematik veranschaulicht werden soll.

Sowohl im Gedächtnis des Objekts 1 als auch im Gedächtnis des Objekts 237 kommt jeweils eine Variable vom abstrakten Datentyp `ADDRESS` vor. Es ist nichts darüber ausgesagt, wie die method_to_processor–Anweisungen aussehen, bei deren Ausführung auf diese Variablen zugegriffen werden muß.

Es wurde angenommen, daß es neben dem abstrakten Datentyp `ADDRESS` auch noch eine abstrakte Klasse `ADDRESS` mit den beiden Unterklassen `ADDRESS_IMPL1` und `ADDRESS_IMPL2` gibt, die sich bezüglich ihrer Aufrufschnittstelle nicht unterscheiden. Objekt 237 sei ein Exemplar der Klasse `ADDRESS_IMPL1`, und Objekt 238 sei ein Exemplar der Klasse `ADDRESS_IMPL2`.

Die Tatsache, daß im Gedächtnis des Objekts 237 nur ein Wert vom Typ `ADDRESS` enthalten ist, bedeutet keineswegs, daß die Methodenmenge M1 bis M5 dem Repertoire der method_to_processor–Anweisungen entsprechen muß, in denen ein Zugriff auf eine Adreßvariable verlangt werden kann.

Die Variable mit dem Namen Other_Address im Gedächtnis des Objekts 1 ist eine Referenzvariable, die aktuell mit dem Wert "Referenz auf das Objekt 237" belegt ist. Referenzen auf Variable gibt es nicht, und man braucht sie auch nicht. Denn auf die Variable My_Addresss sollen nur die Methoden des Objekts 1 und auf die Variable Shared_Address nur die Methoden des Objekts 237 zugreifen können.



Interpretation der Symbole:

Datenvariable, deren Typ
ein abstrakter Datentyp ist

Bild 2 Unterscheidung zwischen Variablen vom Typ ADDRESS und Objekten der Klasse ADDRESS:
Variable vom Typ ADDRESS sind My_Address und Shared_Address;
die Objekte 237 und 238 sind Exemplare zweier nur durch die Implementierung unterschiedenen Unterklassen der abstrakten Klasse ADDRESS.

4.4 Objektreferenzen in Variablen mit abstraktem Datentyp

Bisher wurde noch nichts darüber gesagt, ob es bezüglich der Wertebereiche für Variable mit abstraktem Datentyp irgendwelche Einschränkungen geben soll, d.h. ob grundsätzlich jede Information in einer solchen Variablen untergebracht werden darf oder nicht.

Wenn es denn eine Einschränkung geben sollte, dann könnte diese wohl nur darin bestehen, daß verboten wird, in Variablen, deren Typ ein abstrakter Datentyp ist, Objektreferenzen unterzubringen. Es fällt mir aber kein Grund ein, weshalb eine solche Einschränkung wünschenswert sein könnte.

Bild 3 zeigt ein Beispiel, bei dem Objektreferenzen in drei unterschiedlichen Formen im Gedächtnis des Objekts 1 vorkommen, nämlich in der normalen Referenzvariablen V1, als Arrayelement in der Variablen V2 und schließlich als Information in der Variablen V3, aus der diese Information über eine Funktion der Art

```
REFERENCE_TO_OBJECT_OF_CLASS_xxxx f( V3, sonstige Argumente )
```

extrahierbar sein muß.

4.5 Realisierung abstrakter Datentypen

Bisher wurde nichts darüber ausgesagt, wie die abstrakten Datentypen realisiert werden können. Grundsätzlich gilt: Bei der Realisierung von abstrakten Datentypen verläßt man die Programmebene, in der diese Datentypen verwendet werden – so wie man die Ebene der Maschinenprogramme verläßt, wenn man die Realisierung der einzelnen Maschinenbefehle betrachtet.

Das bedeutet allerdings nicht, daß man auf der tieferliegenden Realisierungsebene zwangsläufig eine völlig andere Programmiersprache vorfinden muß. Man denke beispielsweise an die Realisierung des Befehls zur Gleitkommamultiplikation. In vielen Prozessoren ist dieser Befehl in der Hardware realisiert, also in einer Sprachwelt, die sich von der Maschinensprache des Prozessors grundlegend unterscheidet; in etlichen älteren Prozessoren aber mußte dieser Befehl noch durch Software realisiert werden, also in der gleichen Sprachwelt, in der dieser Befehl verwendet werden sollte. Wichtig ist lediglich, daß es dem Verwender der neuen Elemente – Datentypen oder Befehle – verborgen bleiben kann, wie diese Elemente realisiert wurden.

In dem Beispiel in Bild 4 wurde angenommen, daß die Gedächtnisvariable V4 des Objekts 1 der Ebene 0 von einem abstrakten Datentyp sei, der auf der Ebene –1 realisiert wurde, ohne daß dabei die Sprachwelt gewechselt wurde.

Es ist ganz wichtig zu erkennen, daß es neben den selbstverständlichen Referenzen zwischen Objekten einer Ebene nur noch Referenzen von unten nach oben, nicht aber von oben nach unten geben darf. Denn oben soll kein Wissen über die Existenz von Objekten auf der unteren Ebene vorhanden sein. Die Existenz von Objekten auf der unteren Ebene ergibt sich nämlich erst aufgrund von Entscheidungen des Entwicklers, der die abstrakten Datentypen für die obere Ebene realisiert – und die Geheimhaltung dieser Entscheidungen macht ja gerade den Vorteil der abstrakten Datentypen aus.

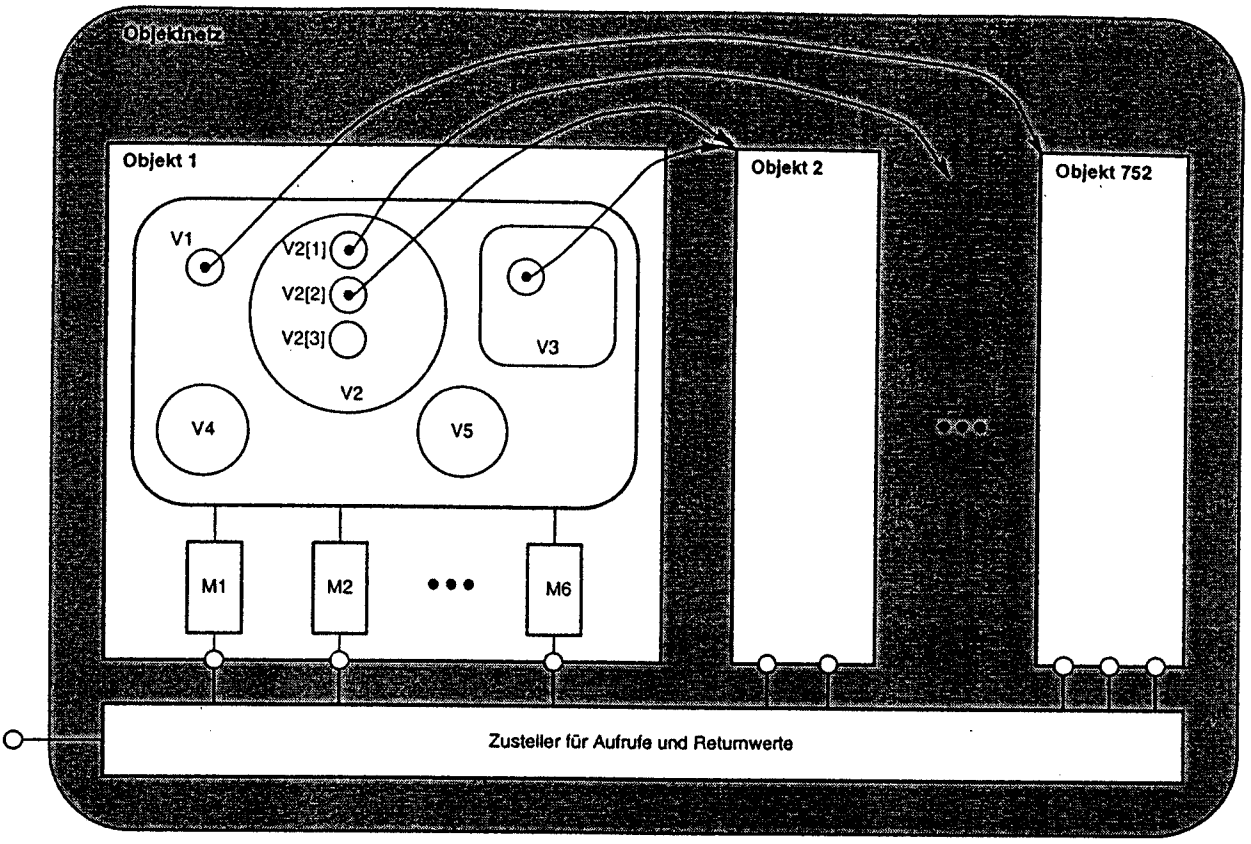


Bild 3 Verschiedene Möglichkeiten, Referenzen im Objektgedächtnis unterzubringen

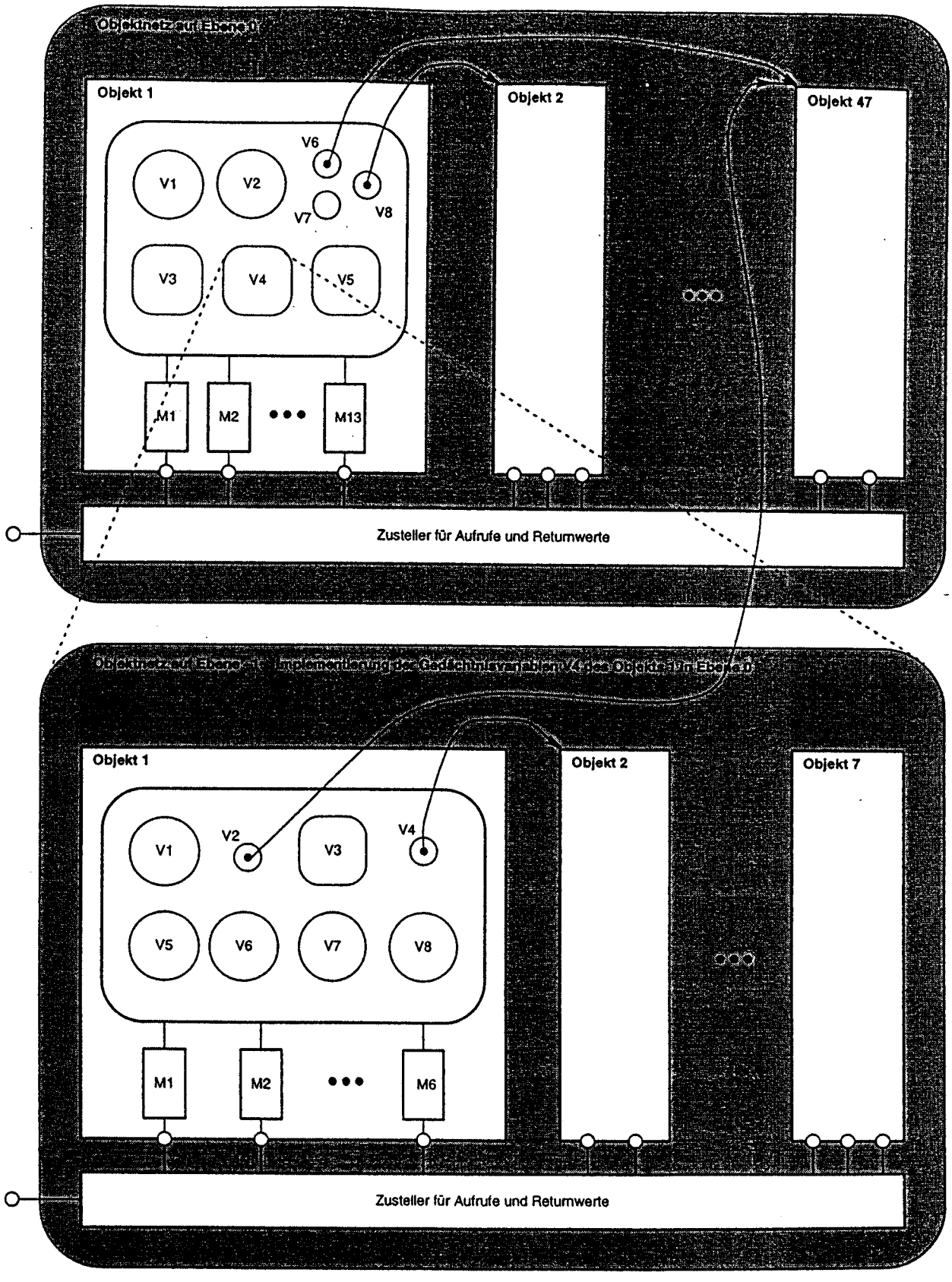


Bild 4 Trennung der Ebenen für Verwendung und Realisierung abstrakter Datentypen

5. Nachträgliche Definition der verwendeten Grundbegriffe

- Eine *Variable* ist ein Ort für Werte, wobei an diesem Ort zu jedem Zeitpunkt entweder ganau ein Wert oder kein Wert definiert ist. Die zu verschiedenen Zeiten an dem Ort vorkommenden Werte dürfen unterschiedlich sein.
  Man kann Speichervariable für ruhende Werte und Kanalvariable für fließende Werte unterscheiden.

- Ein *Wert* ist eine Information, für die es "wertetypische Verwendungen" gibt.
  Für jeden beliebigen Wert sind zumindest die beiden Standardverwendungsarten
  - Kopieren an einen Zielort
  - Vergleichen mit einem Wert von einem anderen Ort

  zugelassen.
  Ein Wert ist immer etwas passives, mit dem etwas getan werden kann.

- Ein *Typ* ist ein Wertebereich für Variable. Er legt fest, mit welchen Werten eine Variable potentiell belegt werden kann und welche Verwendungsarten es für diese Werte gibt.

- Ein *Objekt* ist eine Funktionseinheit mit einem Gedächtnis, das in Gedächtnisvariable gegliedert ist, und mit Methoden, die der Erledigung von object_to_object–Anweisungen dienen.
  Ein Objekt ist immer etwas aktives, das etwas tun kann.

- Eine *Klasse* ist eine Beschreibung der gemeinsamen Merkmale aller Objekte, die dieser Klasse zugeordnet werden sollen.

- Eine *Referenzvariable* ist eine Variable, die nur Werte enthalten kann, die nur als Referenz auf ein Objekt verwendet werden können.
  Die Kennzeichnung einer Variable als Referenzvariable stellt bereits eine Typzuordnung dar. Der Typ der Variable kann noch weiter eingeschränkt werden durch Angabe der Klasse der Objekte, auf die der jeweilige Wert der Variable verweisen kann.
  Referenzen auf Werte gibt es nicht.

- Eine *Funktion* ist eine eindeutige Abbildung einer Menge von Typtupeln auf eine Menge von Typtupeln.

- Eine *Operation* ist eine Änderung der Wertebelegung von Variablen.

- Eine *Anweisung* ist die Beschreibung einer gewünschten Operation. Die beteiligten Variablen werden., soweit sie nicht aufgrund von Vereinbarungen implizit festliegen, durch Namen identifiziert. In der Anweisung kann die Berechnung von Funktionen implizit oder explizit verlangt werden.