7.2 Objektorientierte Programmierung

7.2.1 Modellierungs- und Realisierungsebene

7.2.1.1 Entwicklungsschritte

Bei der SW-Entwicklung werden verschiedene Schritte (oft Phasen genannt) unterschieden. Die wesentlichen Phasen sind Analyse/Modellierung und Realisierung. In der Analyse/Modellierungsphase wird aus den funktionalen und sonstigen Anforderungen ein Modell des zu entwickelnden Systems erstellt. Zur Darstellung dieses Modells werden hauptsächlich grafische Hilfsmittel (Diagramme unterschiedlicher Art) verwendet.

Die eigentliche konstruktive Realisierung der Lösung besteht im wesentlichen aus der Entwurfs- und der Implementierungsphase. In der Entwurfsphase wird die Lösung in ihrer statischen und dynamischen Struktur erarbeitet und festgelegt, während die Ausarbeitung zu einem lauffähigen Programm in der Implementierungsphase erfolgt. Es hat sich eingebürgert, auf beiden Ebenen unterschiedliche Notationen zur Formulierung der Lösung zu verwenden. Die Abhängigkeit zwischen diesen beiden Ebenen ist so, daß in der Implementierungsphase die in der Entwurfsphase erarbeiteten Strukturen durch die auf der Implementierungsebene verfügbaren Strukturierungselemente und Strukturierungsmöglichkeiten zu realisieren sind. Diese Umsetzung ist leichter durchführbar, wenn die auf beiden Ebenen verwendeten Grundkonstruktionen ähnliche Strukturen haben. Letzten Endes hat die Implementierungsebene eine entscheidende Bedeutung, denn es sind die Programme, welche die Rechner steuern. Daher ist es wichtig, daß Modellierer und Entwerfer auch die Grundprinzipien der Grundelemente der Implementierungsebene gut kennen. Dies ist insbesondere bei einer neuen Technik wie der Objektorientierung wichtig.

Die Entwicklung der Konzepte zur SW-Strukturierung erfolgte bisher so, daß neue Konzepte fast immer zuerst auf der Implementierungsebene erfunden wurden und dann anschließend auch auf der Entwurfs- und Modellierungsebene eingesetzt wurden. So ist es auch bei der Erfindung der Konzepte der Objektorientierung gewesen.

7.2.1.2 Modellierungsebene

Auf der Modellierungsebene wird sehr viel mit grafischen Darstellungen wie z.B. Datenfluß- oder ER-Diagrammen gearbeitet (s. II-3.3). Zum Teil werden auch Diagrammformen eingesetzt, die auch auf der Entwurfsebene zum Einsatz kommen. Oft werden auch ad-hoc Diagrammformen ohne festgelegte Syntax verwendet, da es darum geht, die unterschiedlichsten Sachverhalte der Aufgabenstellung überhaupt zu Papier zu bringen. Wenn eine gute Kenntnis der Grundkonstruktionen der Realisierungsebene vorliegt, dann werden aber auch solche ad-hoc Darstellungen so strukturiert sein, daß sie sich leicht durch

Programme realisieren lassen. Die Diagrammformen der Modellierungsebene haben zum Teil auch informellen Charakter, da sie nicht direkt ausführbar sind. Wenn Werkzeuge zur Unterstützung des Zeichnens solcher Diagramme verwendet werden (CASE-Werkzeuge), dann ist dadurch in der Regel eine Syntax für die Diagramme festgelegt (s. z.B. Abb. II-3.3-4); die Semantik ist aber dabei manchmal nicht so präzise festgelegt wie auf der Realisierungsebene.

7.2.1.3 Realisierungsebene

Neben den Diagrammen der Entwurfsebene spielen in der Implementierungsphase programmiersprachliche Formulierungen eine wichtige Rolle. In Programmiersprachen müssen Syntax und Semantik (= Wirkung bei der Ausführung) präzise festgelegt sein, da die Ausführung der Programme ohne weiteres Zutun des Menschen durch den Rechner erfolgt. Eine ggf. zuerst erfolgende Übersetzung in Maschinencode steht dieser Feststellung nicht entgegen, da auch diese Übersetzung ohne weiteres Zutun des Menschen durch einen Automaten (hier den Compiler) erfolgt (s. I-7.1).

Die Darstellungsform ist auf der Implementierungsebene in aller Regel die Textform. Als Ausdrucks- und Strukturierungshilfsmittel stehen nur die Grundkonstruktionen der jeweils verwendeten Programmiersprache zur Verfügung.

7.2.2 Grundlagen der Objektorientierung

7.2.2.1 Entstehung der Objektorientierung

Die Objektorientierung ist eine Fortentwicklung der Programmodularisierung [s. I-7.1, II-3.3.3]. Beide tragen der Beobachtung Rechnung, daß der typische Baustein eines DV-Systems eine Kombination von Daten und Operationen zur Manipulation dieser Daten ist. Abbildung I-7.2-1 gibt einige Beispiele für diese Beobachtung.

Rechner	=	Daten / Dateien + Operationen
Datenbank	=	Daten (Inhalt) + Operationen
Betriebssystem	=	Tabellen + Operationen
Konto	=	Kontendaten + Operationen

Abb. I-7.2-1 Bausteine von DV-Systemen

Aus Benutzersicht besteht ein Rechner im wesentlichen aus Datenbeständen, die mittels Anweisungen oder Kommandos manipuliert werden können. Die Datenbestände können in unterschiedlicher Form gespeichert sein: z.B. in Dateien oder in Datenbanken. Ähnlich besteht auch eine Datenbank aus einem Datenbestand - dem Datenbankinhalt - und Operationen, um Daten einzufügen oder um den Inhalt abzufragen. Ein Betriebssystem ist im wesentlichen eine Menge von Tabellen - z.B. Dateiverzeichnisse - und einer Menge von Operationen, welche die Tabellen manipulieren. Und auch eine kleinere Einheit wie ein Konto besteht aus Daten - wie z.B. Kontoinhaber und Kontostand - und Operationen zur Bearbeitung dieser Daten.

Die Beobachtung, daß gewisse Bausteine oder Systeme eine Kombination von Daten und Operationen sind, ist nicht auf DV-technische Bausteine beschränkt. Auch Organisationseinheiten können so gesehen werden: die Personalabteilung verwaltet die Personalakten und führt darauf Operationen aus wie z.B. Höhergruppierung, Einstellen oder Versetzen. Diese Sicht auf eine Personalabteilung ist in Abb. I-7.2-2 in einer für die Objektorientierung typischen Symbolik dargestellt. Die Kästchen, welche die drei Operationen darstellen ragen aus dem Kästchen, welches die gesamte Personalabteilung darstellt, heraus. Dies soll zum Ausdruck bringen, daß diese Operationen von anderen Objekten angesprochen werden können. Die Ellipse, welche den Datenbestand (Akten) darstellt, befindet sich im Gegensatz dazu ganz im Innern des Objektes „Personalabteilung". Dies soll zum Ausdruck bringen, daß ein direkter Zugriff darauf von außen nicht möglich ist und wird häufig als Datenkapselung bezeichnet.

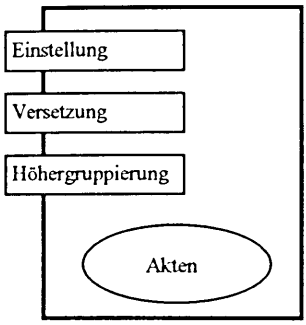


Abb. I-7.2-2 Personalabteilung als Objekt

Ein anderes Beispiel ist die Abteilung Einkauf / Bestellwesen: sie verwaltet die dazugehörigen Unterlagen und bietet nach außen Dienste an wie „Bestellung durchführen", „Statistiken liefern", „Lieferung entgegennehmen" usw.

Man könnte nun vermuten, daß bei der elektronischen Datenspeicherung und beim Trend zu firmenweit vernetzten Datenbeständen die Daten nicht mehr eindeutig den Organisationseinheiten zuzuordnen sind. Um Ordnung und Konsistenz im Gesamtdatenbestand einer Firma zu halten, müssen für Teildatenbestände klare Verantwortlichkeiten festgelegt werden, die sich in entsprechenden Zugriffs- und Manipulationsrechten niederschlagen. Dabei gibt es z.B. in aller Regel auch den Begriff des Eigentümers. Die Teildatenbestände einer Firma können häufig auch nicht so intensiv integriert werden, wie dies rein technisch möglich wäre: z.B. sind personenbezogene Daten, wie sie die Personalabteilung verwaltet, gegen Zugriffe besonders zu schützen, was z.B. dadurch geschehen kann, daß man sie technisch von den anderen Datenbeständen deutlich trennt. Das kann soweit gehen, daß ein solch kritischer Datenbestand auf weitgehend isolierten Rechnern gehalten wird.

Eine Folge aus der Verwendung von Bausteinen, die eine Kombination von Daten und Operationen sind, ist, daß bei Modellierung und Entwurf mit objektorientierten Notationen Funktions- und Datenmodellierung (s. II-3.4, II-3.3) weitgehend integriert erfolgen und nicht so stark getrennt werden, wie bei der herkömmlichen Vorgehensweise.

7.2.2.2 Charakteristika der Strukturierten Software

Früher hat man Daten und Operationen, die im Rechner auf der Prozessorebene und auf der Ebene des Programmierens im Kleinen deutlich getrennt sind, auch auf der Ebene des Programmierens im Großen, d.h. der Ebene der Software-Architektur stark getrennt. Diese Trennung ist z.B. in der Methode der Strukturierten Analyse und des Strukturierten Entwurfs stark ausgeprägt, wo im Datenflußdiagramm (DFD) Prozesse / Funktionen / Operationen und Daten(speicher) unterschieden werden. COBOL unterscheidet z.B. die Programmteile zur Definition der Daten (DATA DIVISION) und den Teil, in welchem die Aktionen formuliert werden (PROCEDURE DIVISION). Zur Beschreibung der Strukturierung von Daten werden häufig Entity-Relationship-Diagramme (ERD) verwendet. Im folgenden wird allgemein von Prozeduren und Daten gesprochen.

Ein Programm oder ein Programmsystem besteht aus einer Menge von Grundbausteinen, die durch verschiedene Beziehungen miteinander verknüpft sind und dadurch einen Wirkungszusammenhang bilden. Folgende strukturellen Beziehungen sind möglich:

- Prozedur P1 benutzt Prozedur P2 (Aufruf von P2 in P1; diese Benutzung ist in der Regel mit einem Datenaustausch zwischen P1 und P2 verbunden)
- Prozedur P2 ist in Prozedur P1 enthalten (hierarchische Strukturierung)
- Prozedur P benutzt Daten D (lesende und schreibende Benutzung)
- Daten D sind in Prozedur P enthalten (und werden auch von P benutzt)
- Datenbestand D1 enthält Datenbestand D2 als Komponente
- Datenbestand D1 verweist auf Datenbestand D2
- Die Daten D1, ..., Dn stehen in der Relation R

Beispiele für Diagramme aus dem Bereich der Strukturierten Software sind in den Abb. II-3.4-6 (DFD) und II-3.3-10 (ERD) enthalten.

7.2.2.3 Charakteristika der Objektorientierten Software

Zur Gestaltung einer objektorientierten SW-Struktur gibt es nun zwei weitere Arten von Grundbausteinen: das Objekt (Modul) und den Objekttyp (Klasse). Ein Objekt ist eine Kombination von Daten und Operationen und ist somit ein Grundbaustein im Sinne von Abb. I-7.2-1. Die Module der modularen SW-Strukturierung (Ada, Modula) sind ebenfalls Kombinationen von Daten und Operationen. In COBOL kommen Programme und Run-Units den Modulen und Objekten am nächsten. Aus heutiger Sicht sind die zwei getrennt entstandenen Dinge „Objekt" und „Modul" im wesentlichen das Gleiche.

Der Objekttyp, der häufig auch als Klasse bezeichnet wird, ist ein Typ im Sinne der typisierten Programmiersprachen (z.B. Ada, Pascal, ggf. zukünftig in SQL3 und COBOL). In der betriebswirtschaftlichen Datenverarbeitung wurde bisher das Konzept des (Daten-) Typs mehr implizit verwendet und nicht so klar zwischen der Definition eines Typs T und eines Datenobjektes, welches den Typ

T hat, unterschieden. Da in objektorientierten SW-Strukturen Objekttypen als Bausteine der statischen Struktur auftreten, muß das Typkonzept nun als eigenständiges Konzept mit betrachtet werden und in seinen technischen Eigenschaften beim SW-Entwurf berücksichtigt werden.

Zur Strukturierung objektorientierter SW gibt es folgende Arten von Bausteinen:

- Objekt / Modul
- Objekttyp
- Prozedur

Dabei kann der klassische Datenbestand als Grenzfall eines Objektes / Moduls gesehen werden, so daß sich bezüglich der Bausteinarten eine Hinzunahme der beiden neuen Arten Objekt und Objekttyp ergibt. Zwischen diesen Bausteinarten gibt es nun noch mehr Beziehungen als oben bei den zwei Bausteinarten der strukturierten Programmierung. Darauf wird weiter unten in Abschnitt 7.2.3 eingegangen. Auf der Entwurfsebene wurden noch weitere Strukturierungselemente eingeführt, die auch zu weiteren Beziehungen führen. Darauf wird in Abschnitt 7.2.4 eingegangen.

7.2.3 Objektorientierung auf der Realisierungsebene

Da es sehr unterschiedliche Programmiersprachen gibt, welche die Objektorientierung unterstützen, wird in diesem Abschnitt zuerst eine neutrale Schreibweise verwendet und dann an einigen Beispielen gezeigt, wie gewisse Konstrukte in einzelnen konkreten Programmiersprachen aussehen.

7.2.3.1 Objekt

Ein Objekt ist eine Zusammenfassung / Aggregierung von Daten und Operationen:

```
VAR Konto1 :   OBJECT
                   TYPE BetragsTyp  =  RANGE 0.0 .. 999_999.99;
                   TYPE KontostandsTyp  =  RANGE 0.0 .. 999_999.99;
                   PROCEDURE Gutschrift (Betrag: IN BetragsTyp);
                   PROCEDURE Lastschrift (Betrag: IN BetragsTyp);
                   FUNCTION Kontostand () RETURN KontostandsTyp;
                   FUNCTION KontoInhaber () RETURN NamensTyp;
               INTERNAL
                   VAR Akt_Kontostand : KontostandsTyp  INIT := 0.0;
                   VAR Akt_KontoInhaber:  NamensTyp := "Egon Müller GmbH";
               END OBJECT;
```

Abb. I-7.2-3 Definition des Objektes Konto1

Die in Abb. I-7.2-3 definierte Variable Konto1 besteht aus einigen Komponenten, welche zur Darstellung eines Kontos erforderlich sind. Diese Komponenten sind deutlich sichtbar zu einer Einheit zusammengefaßt: OBJECT ... END OBJECT. Die Komponenten sind von unterschiedlicher Art: ein Datentyp (BetragsTyp) zur Festlegung des Wertebereiches für den Kontostand, Operationen zur Bearbeitung

des Kontos (Gutschrift, Lastschrift, Kontostand) und zwei Datenkomponenten zur Speicherung des aktuellen Kontostandes (Akt_Kontostand) und des Kontoinhabers (Akt_KontoInhaber). Der genaue Typ von Akt_KontoInhaber ist hier nicht beschrieben. Der aktuelle Kontostand ist mit dem Wert 0.0 initialisiert und der Kontoinhaber mit „Egon Müller GmbH".

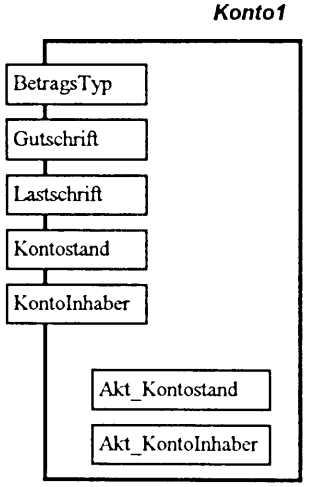


Abb. I-7.2-4 Objekt „Konto1"

In Abb. I-7.2-4 ist das Objekt Konto1 grafisch veranschaulicht analog zur Darstellung in Abb. I-7.2-2. Wie man sieht, entsprechen sich textuelle und grafische Darstellung weitgehend.

Das Objekt Konto1 kann nun folgendermaßen manipuliert werden:

```
Konto1.Gutschrift (2_350.75);
Konto1.Lastschrift (225.30);
IF Konto1.Kontostand() > 1_000.00 THEN . . .
```

Die Manipulation erfolgt dadurch, daß die in der Definition von Konto1 als Komponenten definierten Operationen auf Konto1 angewendet werden. Das generelle Schema einer solchen Manipulation ist:

```
Objektidentifikation . Operationsidentifikation (Parameter)
```

Im obigen Beispiel wird auf Konto1 zuerst eine Gutschrift mit dem Betrag 2_350.75 ausgeübt und dann eine Lastschrift mit dem Betrag 225.30. Der Kontostand beträgt dann also 2_125.45. Die anschließende Abfrage in der Fallunterscheidung „IF Konto1.Konstostand() > 1_000.00" ist daher erfüllt, und es wird die Aktion nach THEN ausgeführt.

7.2.3.2 Öffentliche und interne Komponenten

Betrachtet man die Definition von Konto1 in Abb. I-7.2-2 genauer, dann stellt man fest, daß die Komponenten von Konto1 in zwei Abschnitten notiert sind: zwischen OBJECT und INTERNAL und zwischen INTERNAL und END OBJECT. Der erste Abschnitt wird als öffentlicher Abschnitt bezeichnet und infolgedessen die darin notierten Komponenten als öffentliche Komponenten. Im Beispiel sind dies der Typ BetragsTyp und die Prozeduren Gutschrift, Lastschrift, Kontostand und KontoInhaber. Im Programmtext außerhalb der Definition von Konto1 können nur die öffentlichen Komponenten verwendet werden. Die Benutzung der Prozeduren wurde bereits gezeigt. Der Datentyp BetragsTyp kann z.B. folgendermaßen verwendet werden:

```
VAR Kaufpreis : Konto1.BetragsTyp;
Kaufpreis := 425.30;
Konto1.Lastschrift(Kaufpreis);
```

Die Komponenten im zweiten Abschnitt werden als *interne Komponenten* bezeichnet. Sie sind im Programmtext außerhalb der Variablendefinition nicht ansprechbar, d.h. die folgende Anweisung ist fehlerhaft:

```
Konto1.Akt_Kontostand := 10.0;  – Fehler !!!!
```

und wird daher vom Compiler zurückgewiesen. Diese internen Komponenten können nur intern in der Definition von Konto1 verwendet werden, und daher rührt auch die Bezeichnung interne Komponenten. In Abb. I-7.2-3 ist allerdings keine Benutzung zu erkennen. Dies liegt daran, daß lediglich die Spezifikation des Objektes Konto1 dargestellt ist, welche im wesentlichen die Information enthält, die zur Benutzung und Manipulation des Objektes benötigt wird. Die dargestellte Spezifikation muß noch durch einen Implementierungsteil ergänzt werden, der häufig auch als Rumpf bezeichnet wird.

7.2.3.3 Objektrumpf und Datenkapselung

Ein Rumpf für Konto1 ist in Abb. I-7.2-5 dargestellt.

```
VAR Konto1:   OBJECT BODY
                  PROCEDURE Gutschrift (Betrag: IN BetragsTyp) IS
                      Akt_Kontostand := Akt_Kontostand + Betrag;
                  END Gutschrift;
                  PROCEDURE Lastschrift (Betrag: IN BetragsTyp) IS
                      Akt_Kontostand := Akt_Kontostand - Betrag;
                  END Lastschrift;
                  FUNCTION Kontostand () RETURN KontostandsTyp IS
                      RETURN Akt_Kontostand;
                  END Kontostand;
                  FUNCTION KontoInhaber () RETURN NamensTyp IS
                      RETURN Akt_KontoInhaber;
                  END KontoInhaber;
              END OBJECT;
```

Abb. I-7.2-5 Rumpf von Konto1

Die internen Datenkomponenten können nur über die in der öffentlichen Schnittstelle exportierten Operationen (Prozeduren und Funktionen) manipuliert werden. Aus Abb. I-7.2-5 geht hervor, daß Akt_Kontostand durch die Operationen Gutschrift und Lastschrift verändert werden kann. Die Komponente Akt_KontoInhaber kann aber nur mit der Operation KontoInhaber abgefragt werden und hat stets den in der Initialisierung bei Erzeugung der Variablen Konto1 festgelegten Wert. Die so bewirkte Abschottung interner Datenkomponenten nach außen wird auch als Datenkapselung bezeichnet und ist ein wichtiges Hilfsmittel zur Erstellung korrekter Programme.

7.2.3.4 Objekttyp / Klasse

Benötigt man weitere Konten, dann kann man weitere Variablen dieser Art definieren. In Abb. I-7.2-6 ist die Variable Konto2 definiert. Es ist nur der Spezifikationsteil gezeigt, da der Rumpf textuell identisch ist zu dem von Konto1 (s. Abb. I-7.2-5).

```
VAR Konto2  :  OBJECT
                   TYPE BetragsTyp  =  RANGE 0.0 .. 999_999.99;
                   TYPE KontostandsTyp = RANGE 0.0 .. 999_999.99;
                   PROCEDURE Gutschrift (Betrag: IN BetragsTyp);
                   PROCEDURE Lastschrift (Betrag: IN BetragsTyp);
                   FUNCTION Kontostand () RETURN KontobestandsTyp;
                   FUNCTION KontoInhaber () RETURN NamensTyp;
               INTERNAL
                   VAR Akt_Kontostand : KontostandsTyp  INIT := 10_000.0;
                   VAR Akt_Inhaber:  NamensTyp INIT := "EDV Besch 1";
               END OBJECT;
```

Abb. I-7.2-6 Definition der Variablen Konto2

Eine Manipulation von Konto2 ist analog wie bei Konto1 möglich:

```
Konto2.Gutschrift (50_000.00);
Konto2.Lastschrift (5_350.00);
```

Benötigt man viele Konten, dann ist das hier gewählte Vorgehen, jedesmal eine solche Variable zu definieren, umständlich, da jedesmal die gesamte Beschreibung (Spezifikation und Rumpf) zu wiederholen ist, obwohl sich die verschiedenen Variablen nur im Variablennamen (Konto1, Konto2) und den Intialwerten für Kontostand und Kontoinhaber unterscheiden. Dieses Problem kann auf zweierlei Art gelöst werden:

- mit einer OCCURS-Klausel wie in COBOL
- mit einer Typdefinition

Die OCCURS-Klausel von COBOL stellt nur eine eingeschränkte Lösung dar, da sie es nur erlaubt, mehrere Exemplare von ähnlichen Variablen an einer Stelle im Programm zu erzeugen, nämlich dort, wo die OCCURS-Klausel steht. Im Rahmen der Fortentwicklung ist aber im Gespräch, auch in COBOL Typdefinitionen einzuführen [BS 95].

Der zweite Weg, eine Typdefinition, wird in vielen Programmiersprachen verwendet und ist auch der in der Objektorientierung üblicherweise benutzte Ansatz. Letzten Endes ist dieser Weg nicht spezifisch für Programmiersprachen oder für Software. Es handelt sich um die in vielen Gebieten bewährte Technik, die Komplexität einer Beschreibung dadurch zu reduzieren, daß Teile durch Namen abgekürzt werden, und dann diese Namen anstelle der jeweiligen Teilbeschreibung verwendet werden. In Abb. I-7.2-7 ist nun eine Typdefinition für den Typ KontoTyp angegeben.

```
TYPE KontoTyp  = OBJECT
                   TYPE BetragsTyp  = RANGE 0.0 .. 999_999.99;
                   TYPE KontostandsTyp  = RANGE 0.0 .. 999_999.99;
                   PROCEDURE Gutschrift (Betrag: IN BetragsTyp);
                   PROCEDURE Lastschrift (Betrag: IN BetragsTyp);
                   FUNCTION Kontostand ( ) RETURN KontostandsTyp;
                   FUNCTION KontoInhaber ( ) RETURN NamensTyp;
                   CONSTRUCTOR KontoTyp (Inhaber: IN NamensTyp;
                                         Anfangsstand: IN KontostandsTyp);
                 INTERNAL
                   VAR Akt_Kontostand : KontostandsTyp;
                   VAR Akt_KontoInhaber:  NamensTyp;
                 END OBJECT;

TYPE KontoTyp  =  OBJECT  BODY
                   PROCEDURE Gutschrift (Betrag: IN BetragsTyp) IS
                     Akt_Kontostand := Akt_Kontostand + Betrag;
                   END Gutschrift;
                   PROCEDURE Lastschrift (Betrag: IN BetragsTyp) IS
                     Akt_Kontostand := Akt_Kontostand - Betrag;
                   END Lastschrift;
                   FUNCTION Kontostand ( ) RETURN KontostandsTyp IS
                     RETURN Akt_Kontostand;
                   END Kontostand;
                   FUNCTION KontoInhaber ( ) RETURN NamensTyp IS
                     RETURN Akt_Inhaber ;
                   END KontoInhaber;
                   CONSTRUCTOR KontoTyp (Inhaber: IN NamensTyp;
                                         Anfangsstand: IN KontostandsTyp)  IS
                     Akt_Kontostand := Anfangsstand;
                     Akt_KontoInhaber := Inhaber;
                   END KontoTyp;
                 END OBJECT;
```

Abb. I-7.2-7 Definition des Typs KontoTyp

Ein Typ von der Art KontoTyp kann naheliegenderweise als *Objekttyp* bezeichnet werden. Diese Bezeichnung findet sich auch bereits ab und zu, z.B. bei Borland-Pascal [Bor 93] und bei CORBA [OMG 91]. Häufig findet sich auch noch der Ausdruck *Klasse* für einen solchen Typ. Da der Ausdruck Klasse aber auch für Verwirrung sorgen kann [Win 92], wird hier der sinnfälligere Ausdruck *Typ* bzw. *Objekttyp* verwendet.

Die Definition des Typs KontoTyp ist weitgehend identisch mit den Definitionen der Variablen Konto1 und Konto2. Der wesentliche Unterschied besteht darin, daß der Typ KontoTyp eine weitere Operation mit dem Namen „KontoTyp" enthält, die mit dem Schlüsselwort CONSTRUCTOR gekennzeichnet ist. Dies besagt, daß diese Operation automatisch bei der Vereinbarung einer Variablen V vom Typ KontoTyp auf diese Variable V angewendet wird. Der Hauptverwendungszweck dieser Konstruktoren ist die Initialisierung der neu geschaffenen Variablen. Da Konstruktoren auch Parameter haben können, ist damit eine sehr flexible Initialisierung möglich. Ein solcher Konstruktor darf meist nur in dieser Rolle innerhalb einer Variablendeklaration verwendet werden. In Abb. I-7.2-7 liegt eine Entsprechung zwischen den Parametern des Konstruktors KontoTyp und den Datenkomponenten des Typs KontoTyp vor.

Dies ist zufällig in diesem Beispiel so, darf aber nicht als eine allgemeine Regel interpretiert werden. Als Pendant zu den Konstruktoren ist es in der Regel auch möglich Destruktoren zu definieren, die automatisch beim Vernichten einer Variablen ausgeführt werden. Im vorliegenden Beispiel ist kein Destruktor formuliert.

Der Typ KontoTyp kann nun folgendermaßen verwendet werden:

```
VAR Konto1 : KontoTyp (Inhaber => "Egon Müller GmbH", Anfangsstand => 0.0);        (1)
```

Dadurch wird eine Variable Konto1 definiert, die ebenso aufgebaut ist wie die in Abb. I-7.2-3 definierte Variable gleichen Namens. Das Auftreten des Namens „KontoTyp" in (1) hat nun zwei Bedeutungen:

(a) einmal sagt es aus, daß Konto1 den Typ KontoTyp hat, d.h. entsprechend der Struktur des Typs KontoTyp aufgebaut ist,

(b) andererseits ist „KontoTyp(Inhaber => "Egon Müller GmbH", Anfangsstand => 0.0)" der Aufruf des Konstruktors. Dies bewirkt hier die Initialisierung der internen Datenkomponenten der Variablen Konto1.

Entsprechend kann die Variable Konto2 definiert werden:

```
VAR Konto2 : KontoTyp (Inhaber => "EDV Besch 1", Anfangsstand => 10_000.0);
```

Auf diese Weise kann man nun beliebig viele andere Konten definieren, deren Gemeinsamkeiten in der Typdefinition des Typs KontoTyp festgelegt sind. Die Manipulationen der Variablen Konto1 und Konto2 erfolgen dann ganz genauso wie bereits weiter oben gezeigt.

Statt von Variablen eines Objekttyps wird in der Literatur auch von der *Instanz* eines Typs oder der Instanz einer Klasse gesprochen. Da in Programmiersprachen, welche Typdefinitionen erlauben, der Ausdruck *Variable* praktisch durchgängig verwendet wird, wird dieser Bezeichnungsweise hier der Vorzug gegeben.

7.2.3.5 Konkrete Programmbeispiele

Zum Abschluß dieses Abschnittes ist das obige Beispiel noch in den Sprachen Borland-Pascal mit Objekten (Abb. I-7.2-8) und C++ (Abb. I-7.2-9) dargestellt.

```
UNIT Konten;
INTERFACE
    TYPE BetragsTyp = 0 .. 999_999;
    TYPE KontostandsTyp = 0 .. 999_999;
    TYPE KontoTyp = OBJECT
        PROCEDURE Gutschrift (Betrag: BetragsTyp);
        PROCEDURE Lastschrift (Betrag: BetragsTyp);
        FUNCTION Kontostand : KontostandsTyp;
        FUNCTION KontoInhaber : NamensTyp;
        CONSTRUCTOR Init (Inhaber: NamensTyp; Anfangsstand: KontostandsTyp);
    PRIVATE
        Akt_Kontostand : KontostandsTyp;
        Akt_KontoInhaber: NamensTyp;
    END;
```

```
IMPLEMENTATION
      PROCEDURE KontoTyp.Gutschrift (Betrag: BetragsTyp);
      BEGIN    Akt_Kontostand := Akt_Kontostand + Betrag;  END ;
      PROCEDURE KontoTyp.Lastschrift (Betrag: BetragsTyp);
      BEGIN    Akt_Kontostand := Akt_Kontostand - Betrag;  END;
      FUNCTION KontoTyp.Kontostand: KontostandsTyp;
      BEGIN    Kontostand := Akt_Kontostand;  END;
      FUNCTION KontoTyp.KontoInhaber: NamensTyp;
      BEGIN    KontoInhaber :=  Akt_Inhaber ;  END;
      CONSTRUCTOR KontoTyp.Init (Inhaber: NamensTyp;  Anfangsstand:  KontostandsTyp);
      BEGIN    Akt_Kontostand := Anfangsstand;
               Akt_KontoInhaber := Inhaber;
      END;
END.
```

Abb. I-7.2-8 Definition des Typs KontoTyp in Borland-Pascal

Die Definition und Manipulation der Variablen Konto1 und Konto2 erfolgt so:

```
VAR  Konto1, Konto2 : Konten.KontoTyp;
Konto1.Init ("Egon Müller GmbH", 0);
Konto2.Init ("EDV Besch 1", 10000);
Konto1.Gutschrift (2350);
Konto1.Lastschrift (225);
IF Konto1.Kontostand > 1000  THEN . . .
```

Die Definition in C++ ist ähnlich strukturiert. Des wesentliche Unterschied ist der, daß der Objekttyp („class") direkt als Baustein auftritt und nicht in einen anderen Baustein eingeschachtelt ist, wie dies in Abb. I-7.2-8 der Fall ist.

```
class KontoTyp {
protected:
      float  Akt_Kontostand;
      NamensTyp  Akt_KontoInhaber ;
public:
      void  Gutschrift (float  Betrag)  {
            if  (Betrag > 0.0  &&  Akt_Kontostand+Betrag <= Max_Float)
                  Akt_Kontostand  = Akt_Kontostand + Betrag;
            else  "melde Fehler";    }
      void  Lastschrift (float  Betrag)  {
            if  (Betrag > 0.0  &&  Akt_Kontostand - Betrag >= 0.0)
                  Akt_Kontostand  = Akt_Kontostand - Betrag;
            else  "melde Fehler";    }
      float  Kontostand ( )  { return  Akt_Kontostand;   }
      NamensTyp  KontoInhaber ( ) { return  Akt_KontoInhaber;  }
      KontoTyp (NamensTyp  Inhaber, float  Anfangsstand)  {
            if  (Betrag >= 0.0 )  Akt_Kontostand = Anfangsstand;
            else  "melde Fehler";  }    }
}
```

Abb. I-7.2-9 Definition des Typs KontoTyp in C++

Vereinbarung und Manipulation der Variablen Konto1 und Konto2 erfolgt folgendermaßen:

```
KontoTyp  Konto1("Egon Müller GmbH", 0.0);
KontoTyp  Konto2 ("EDV Besch 1", 10000.0);
Konto1.Gutschrift (2350.75);
```

```
Konto1.Lastschrift (225.30);
if (Konto1.Kontostand( ) > 1000.0)  ...
```

In den Programmen in Abb. I-7.2-8 und I-7.2.9 sind auch eine Reihe von sprachspezifischen Besonderheiten enthalten. Auf diese soll hier nicht weiter eingegangen werden, es soll dem Leser lediglich ein Eindruck vermittelt werden, wie dieses Beispiel in realen Programmiersprachen aussehen kann.

7.2.3.6 Allgemeinheit des Ansatzes

Die in den vorstehenden Beispielen realisierten Objekte Konto1 und Konto2 sind relativ „klein" und einfach aufgebaut. Generell kann man feststellen, daß sich beliebig „große" und komplexe Objekte auf diese Art und Weise realisieren lassen; z.B. solche wie in Abb. I-7.2-1 erwähnt. Techniken zur Unterstützung der Kooperation unterschiedlicher Anwendungen auf unterschiedlichen Plattformen wie z.B. COM, SOM und CORBA [Adl 95] basieren auch auf dem Objektparadigma.

Der Leser sei noch darauf hingewiesen, daß es manchmal Schwierigkeiten bei der Verwendung von Objekttypen geben kann, und zwar dann, wenn symmetrische Operationen zu realisieren sind, d.h. wenn eine Operation mehrere Objekte des betreffenden Objekttyps als Eingabe verarbeitet (z.B. bei der Multiplikation zweier Matrizen). Wie solche Fälle im Rahmen der Objektorientierung zu behandeln sind, ist derzeit noch Gegenstand der Diskussion unter den Fachleuten.

7.2.3.7 Ableitung / Typerweiterung / Vererbung

Bei der SW-Entwicklung tritt häufig die Situation auf, daß Größen von leicht unterschiedlichem Typ gebraucht werden. Dies kann in der Problemstellung von Anfang an enthalten sein oder sich im Laufe der Zeit bei der Fortentwicklung ergeben. Im Rahmen der Objektorientierung ist ein Mechanismus entwickelt worden, mit welchem man leicht neue Objekttypen definieren kann, die eine Erweiterung bestehender Objekttypen sind.

Wenn z.B. auch Konten benötigt werden, die gesperrt werden können, dann kann dies durch eine Typdefinition wie in Abb. I-7.2-10 erfolgen. Dabei ist zu beachten, daß Abb. I-7.2-10 nur den Spezifikationsteil enthält und auf der Abb. I-7.2-7 aufbaut. Die Sperre soll so wirken, daß die Operationen Gutschrift und Lastschrift auf ein gesperrtes Konto keine Wirkung haben sollen.

```
TYPE KontoMitSperreTyp =
     OBJECT  BASED_ON  KontoTyp                                  -- Ableitungsklausel
        PROCEDURE Gutschrift (Betrag: IN BetragsTyp);             -- Änderung
        PROCEDURE Lastschrift (Betrag: IN BetragsTyp);            -- erforderlich
        PROCEDURE Sperren ();                                     -- zusätzliche
        PROCEDURE  EntSperren ();
        FUNCTION Gesperrt () RETURN Boolean;                      -- Operationen
        CONSTRUCTOR KontoMitSperreTyp (Inhaber: IN NamensTyp;     -- neuer
                                       Anfangsstand: IN KontostandsTyp;
                                       Sperre: IN Boolean);       -- Konstruktor
```

```
    INTERNAL
       VAR  Konto_Gesperrt : Boolean;                    -- zusätzliche Datenkomponente
    END OBJECT;
```

Abb. I-7.2-10 Spezifikationsteil des Typs KontoMitSperreTyp

Die Spezifikation des Typs KontoMitSperreTyp zeigt nun alle Merkmale, die üblicherweise bei der Ableitung neuer Objekttypen von existierenden vorkommen.

7.2.3.8 Ableitungsklausel und Erweiterung

Nach dem Schlüsselwort OBJECT ist die Ableitungsklausel „BASED_ON KontoTyp" angegeben, die festlegt, daß KontoMitSperreTyp von KontoTyp abgeleitet ist. Die Wirkung der Ableitungsklausel ist nun so, daß so getan wird, als seien alle Komponenten, die in KontoTyp definiert sind, auch in KontoMitSperreTyp definiert. Ausgenommen sind hier meist die Konstruktoren und Destruktoren. Man sagt auch, daß KontoMitSperreTyp die Komponenten von KontoTyp übernimmt oder erbt. Dies ist in Abb. I-7.2-11 dargestellt, in welcher die übernommenen Komponenten als Kommentare (beginnend mit „--") eingeblendet sind.

```
TYPE KontoMitSperreTyp  =
     OBJECT  BASED_ON  KontoTyp                                  -- Ableitungsklausel
        -- TYPE BetragsTyp  =  RANGE 0.0 .. 999_999.99;          -- übernommen
        -- TYPE KontostandsTyp  =  RANGE 0.0 .. 999_999.99;      -- übernommen
        PROCEDURE Gutschrift (Betrag: IN BetragsTyp);            -- Änderung
        PROCEDURE Lastschrift (Betrag: IN BetragsTyp);           -- erforderlich
        -- FUNCTION Kontostand ( ) RETURN KontostandsTyp;        -- übernommen
        -- FUNCTION KontoInhaber ( ) RETURN NamensTyp;           -- übernommen
        PROCEDURE Sperren ( );                                   -- zusätzliche
        PROCEDURE  EntSperren ( );
        FUNCTION Gesperrt ( ) RETURN Boolean;                    -- Operationen
        CONSTRUCTOR KontoMitSperreTyp (Inhaber: IN NamensTyp;    -- neuer
                                       Anfangsstand: IN KontostandsTyp;
                                       Sperre: IN Boolean);      -- Konstruktor
     INTERNAL
        -- VAR Akt_Kontostand: KontostandsTyp;                   -- übernommen
        -- VAR Akt_KontoInhaber:  NamensTyp;                     -- übernommen
        VAR  Konto_Gesperrt : Boolean;                   -- zusätzliche Datenkomponente
     END OBJECT;
```

Abb. I-7.2-11 Spezifikationsteil des Typs KontoMitSperreTyp mit übernommenen Komponenten

Vergleicht man Abb. I-7.2-7 und I-7.2-11, dann stellt man fest, daß abgesehen vom Konstruktor der Typ KontoMitSperreTyp eine Erweiterung des Typs KontoTyp ist; dies gilt sowohl für den öffentlichen wie auch für den internen Teil. D.h. sowohl der öffentliche wie auch der interne Teil enthalten hier neue, zusätzliche Komponenten. Daher wird auch oft von einer Typerweiterung gesprochen.

Der Typ KontoTyp wird als Basistyp von KontoMitSperreTyp bezeichnet und umgekehrt KontoMitSperreTyp als ein von KontoTyp abgeleiteter Typ. Man

spricht auch davon, daß beide Typen in der Ableitungsbeziehung zueinander stehen. Diese Bezeichnungen werden auch verwendet, wenn die Ableitung über mehrere Stufen erfolgt.

Man spricht auch davon, daß Basistyp und abgeleiteter Typ in der *Ableitungsbeziehung* stehen (auch Vererbungsbeziehung genannt). Diese Ableitungsbeziehung ist eine der wichtigen Beziehungen zur Strukturierung objektorientierter SW. Aus der Darstellung der Ableitung (s.o. Abb. I-7.2-11) geht hervor, daß die Ableitungsbeziehung eine enge Kopplung zwischen Basistyp und abgeleitetem Typ bewirkt, da der abgeleitete Typ alle Komponenten des Basistyps (externe und interne) übernimmt. Insbesondere die Übernahme der Externschnittstelle des Basistyps hat einen großen Einfluß auf den abgeleiteten Typ. Die zweite wichtige Beziehung zwischen Programmbausteinen ist die Benutzungsbeziehung, die weiter unten erläutert wird.

7.2.3.9 Mehrfachableitung

In den oben aufgeführten Beispielen hat ein abgeleiteter Objekttyp stets genau einen Basistyp. In manchen Programmiersprachen und Entwurfsnotationen ist es möglich, daß ein Objekttyp unmittelbar von mehreren Basistypen abgeleitet wird. Dies wird auch als Mehrfachvererbung bezeichnet. Dabei treten jedoch eine Reihe von Komplikationen auf. Daher befindet sich dieses Konzept derzeit noch in der Diskussion und kann noch nicht als abgeklärt angesehen werden.

7.2.3.10 Reimplementierung / Modifikation

Da die Operationen Gutschrift, Lastschrift und Kontostand bei einem gesperrten Konto keine Wirkung haben sollen, müssen ihre Rümpfe modifiziert werden. Daher enthält die Spezifikation in Abb. I-7.2-11 erneut Spezifikationen der entsprechenden beiden Prozeduren. Abb. I-7.2-12 enthält den Rumpf von KontoMitSperreTyp. Dieser Rumpf enthält einerseits die Rümpfe der neuen Operationen und die neuen Rümpfe der beiden modifizierten Operationen. Da man Rümpfe oft auch als Implementierungsteile bezeichnet, werden Prozeduren mit neuen Rümpfen auch als reimplementierte Prozeduren bezeichnet. In der Literatur findet man dafür auch den Ausdruck „redefinierte Prozeduren". Dies ist aber eher mißverständlich, da die Definition, d.h. der Prozedurkopf (Prozedurspezifikation) völlig unverändert ist. Ein neuer, unterschiedlicher Prozedurkopf führt ja eine völlig neue Operation ein.

```
TYPE KontoMitSperreTyp = OBJECT BODY
        PROCEDURE Gutschrift (Betrag: IN BetragsTyp) IS                    -- neue
            IF NOT Gesperrt( )
            THEN Konto.Gutschrift (Betrag);                                -- Rümpfe
            END IF;
        END Gutschrift;                                                    -- für
        PROCEDURE Lastschrift (Betrag: IN BetragsTyp) IS
            IF NOT Gesperrt( )                                             -- Gutschrift
            THEN Konto.Lastschrift(Betrag);                                -- und
            END IF;
        END Lastschrift;                                                   -- Lastschrift
        PROCEDURE Sperren ( ) IS                                           -- Rümpfe für
            Konto_Gesperrt := True;
        END Sperren;
        PROCEDURE EntSperren ( )                                           -- zusätzliche
            Konto_Gesperrt := False;
        END Entsperren;
        FUNCTION Gesperrt ( ) RETURN Boolean IS                            -- Operationen
            RETURN Konto_Gesperrt;
        END Gesperrt;
        CONSTRUCTOR KontoMitSperreTyp (Inhaber: IN NamensTyp;              -- neuer
                                       Anfangsstand: IN KontostandsTyp;
                                       Sperre: IN Boolean);                -- Konstruktor
            Konto.Konto (Inhaber, Anfangsstand);
            Konto_Gesperrt := Sperre;
        END KontoMitSperreTyp;
END OBJECT;
```

Abb. I-7.2-12 Rumpf des Typs KontoMitSperreTyp

Im Rumpf in Abb. I-7.2-12 wird in den Situationen, in welchen es möglich ist, in den reimplementierten Rümpfen auf die Rümpfe der entsprechenden Prozeduren im Basistyp Konto zurückgegriffen. Um Namenskonflikte zu vermeiden, ist dann der Prozedurname mit dem Namen des Basistyps qualifiziert. Beispiele dafür sind: „Konto.Gutschrift (Betrag);“ im Rumpf von Gutschrift und „Konto.Konto (Inhaber, Anfangsstand);“ im Rumpf des neuen Konstruktors KontoMitSperreTyp. Im letzteren Falle ist diese Qualifizierung zwar nicht unbedingt erforderlich, wurde aber der Deutlichkeit wegen auch hier gemacht. Diese Wiederverwendung der Rümpfe des Basistyps hat im allgemeinen den Vorteil, daß Vorgänge nicht unnötigerweise mehrfach programmiert werden müssen. Im vorliegenden Beispiel, welches gegenüber der Praxis stark vereinfacht ist, sind die Rümpfe sehr klein. Bei größeren Rümpfen ist der Vorteil dieser Wiederverwendung natürlich entsprechend größer.

Der Typ KontoMitSperreTyp kann nun ganz entsprechend wie der Typ Konto verwendet werden.

Zusammenfassend können in einem abgeleiteten Typ folgende Modifikationen gegenüber dem Basistyp vorgenommen werden:

- Aufnahme neuer (zusätzlicher) Komponenten
- Reimplementierung von Prozeduren

Weglassen von Komponenten ist normalerweise nicht möglich. Dies hängt mit dem Polymorphismus zusammen, der im nächsten Abschnitt behandelt wird.

7.2.3.11 Basistyp als Zusammenfassung von gemeinsamen Komponenten

Der Ableitungsmechanismus kann auch dadurch zum Einsatz kommen, daß in mehreren existierenden Objekttypen Gemeinsamkeiten festgestellt werden, welche in einem Basistyp zusammengefaßt werden können.

```
TYPE KontoMitSperreTyp
   = OBJECT
      BetragsTyp
      KontostandsTyp
      Gutschrift
      Lastschrift
      Kontostand
      KontoInhaber
      Sperren
      EntSperren
      Gesperrt
      KontoMitSperreTyp
   INTERNAL
      Akt_Kontostand
      Akt_KontoInhaber
      Konto_Gesperrt
   END OBJECT;
```

```
TYPE KontoTyp
   = OBJECT
      BetragsTyp
      KontostandsTyp
      Gutschrift
      Lastschrift
      Kontostand
      KontoInhaber
      KontoTyp
   INTERNAL
      Akt_Kontostand
      Akt_KontoInhaber
   END OBJECT;
```

```
TYPE KontoMBuTyp
   = OBJECT
      BetragsTyp
      KontostandsTyp
      BuchgTyp
      Gutschrift
      Lastschrift
      Kontostand
      KontoInhaber
      DruckeBuchungn
      KontoMBuTyp
   INTERNAL
      Akt_Kontostand
      Akt_KontoInhaber
      Akt_Buchungen
   END OBJECT;
```

Abb. I-7.2-12a KontoTyp, KontoMitSperreTyp und KontoMBuTyp als unabhängige Typen

In den folgenden Beispielen werden die Objekttypen nur schematisch dargestellt, da die Details wie Parameter von Operationen für den darzustellenden Sachverhalt nicht wichtig sind.

In Abb. I-7.2-12a sind die drei Typen KontoTyp, KontoMitSperreTyp und KontoMBuTyp so dargestellt, als seien sie unabhängig voneinander entstanden. KontoTyp und KontoMitSperreTyp seien wie bisher. KontoMBuTyp sei wie KontoTyp und habe die Zusatzfunktion, daß alle Buchungen in der Variablen Akt_Buchungen gespeichert werden. Mit der Operation DruckeBuchungn können diese Buchungen ausgedruckt werden.

Man erkennt nun, daß alle drei Typen eine Reihe von dem Namen nach gleichen Komponenten enthalten. Zur Betonung sind diese gemeinsamen Komponenten in Abb. I-7.2-12a kursiv geschrieben. Bei Inspektion der vollständigen Typdefinitionen würde man feststellen, daß BetragsTyp, KontostandsTyp, Kontostand, KontoInhaber, Akt_Kontostand und Akt_KontoInhaber in beiden Typen identisch sind; Gutschrift und Lastschrift haben zwar den gleichen Prozedurkopf, haben aber unterschiedliche Rümpfe.

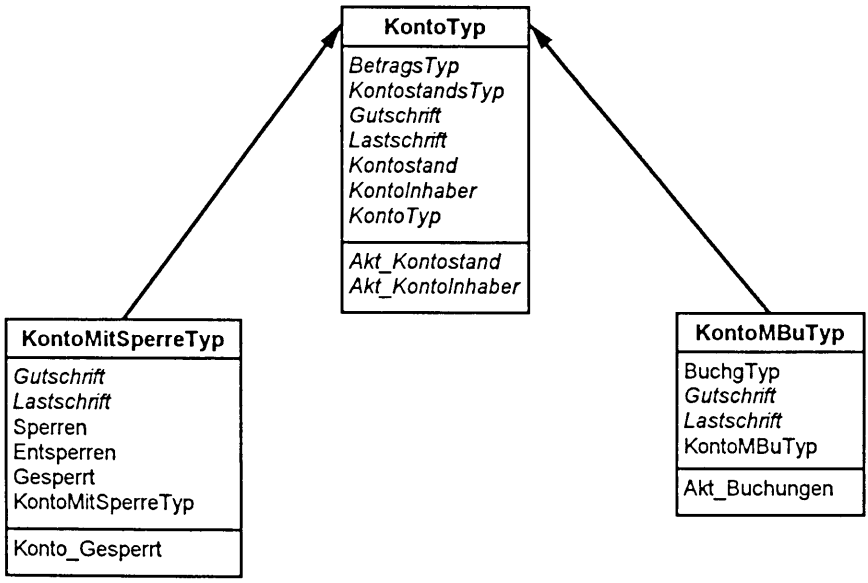


Abb. I-1.7-12b KontoTyp, KontoMitSperreTyp und KontoMBuTyp bei Einsatz der Ableitung

Die Typerweiterung („BASED_ON") kann nun ausgehend von der Situation in Abb. I-7.2-12a so eingesetzt werden, daß ein Basistyp definiert wird, der die gemeinsamen Komponenten enthält und von welchem die ursprünglichen Typen abgeleitet werden. In Abb. I-7.2-12a liegt insofern eine besondere Situation vor, als einer der Typen nur gemeinsame Komponenten enthält. Dies wird im allgemeinen nicht der Fall sein. Nach diesem Herausfaktorisieren der gemeinsamen Komponenten ergibt sich die in Abb. I-7.2-12b dargestellte Situation. Diese Situation tritt in der Praxis ebenfalls häufig auf, da sich bei der Programmentwicklung die genauen Komponenten von Objekten und Objekttypen erst bei der Detaillierung ergeben und entsprechend festgelegt werden.

7.2.3.12 Polymorphismus

In einem Anwendungsfall werden meistens nicht nur zwei einzelne Konten benötigt, wie weiter oben dargestellt, sondern größere Anzahlen. Diese wird man

nicht als einzelne Variablen definieren sondern z.B. als Reihung (Array). Wenn man dies versucht, stößt man aber auf eine Schwierigkeit. Wenn man definiert:

```
CONST Anzahl_Konten = 1000;
KontenA: ARRAY(1..Anzahl_Konten) OF KontoTyp;
```

dann sind alle Elemente der Reihung KontenA vom Typ KontoTyp. Dies ist hinderlich, wenn man sowohl einfache Konten als auch Konten mit Sperrmöglichkeit verwalten will. Der Grund liegt in der in den üblichen Programmiersprachen vorhandenen strengen Typbindung, die im allgemeinen sehr nützlich ist, weil sie es ermöglicht, daß Compiler viele fehlerhafte Konstruktionen erkennen können, wie z.B.

```
KontenA(1).Gutschrift ("ABC");
```

Dies ist fehlerhaft, da die Prozedur Gutschrift einen Ausdruck mit einem Zahlenwert als Parameter verlangt und keine Zeichenkette. Dieser Fehler wird bereits vor Einsatz des Programmes bei der Übersetzung erkannt.

Selbst wenn man Verweise (Pointer) als Elemente der Reihung verwendet, kommt man nicht weiter:

```
KontenB: ARRAY(1..Anzahl_Konten) OF POINTER TO KontoTyp;
```

Auch hier müssen dann alle Elemente vom Typ „POINTER TO KontoTyp" sein.

Wenn man bei den üblichen Typregeln bliebe, könnte man die Idee, in einer Reihung Konten von beiden Arten zu verwalten, nicht realisieren. Daher ist in objektorientierten Sprachen die Typisierungsregel für Verweise folgendermaßen geändert:

```
Verweis_Variable : POINTER TO KontoTyp;
```

Die Variable Verweis_Variable darf nun Verweise auf Variable des Typs KontoTyp *und Verweise auf Variable eines von KontoTyp abgeleiteten Typs* auf nehmen.

Wendet man diese neue Typregel auf die Variable KontenB an, dann ist folgendes möglich:

```
KontenB(1) := NEW KontoTyp ("XYZ AG", 0.0);
KontenB(2) := NEW KontoMitSperreTyp ("ABC GmbH", 10_000.0, False);
KontenB(3) := NEW KontoTyp ("Rela AG", 5_000.0);
```

Dadurch wird eine neue Variable vom Typ KontoTyp erzeugt und mit dem Konstruktor initialisiert und dann ein Verweis auf diese neue Variable in KontenB(1) gespeichert. Dem Reihungselement KontenB(2) wird hingegen ein Verweis auf eine Variable des Typs KontoMitSperreTyp zugewiesen und KontenB(3) wieder ein Verweis auf ein einfaches Konto. Zu einem späteren Zeitpunkt könnte auch KontenB(3) einen Verweis auf ein Konto mit Sperrmöglichkeit aufnehmen:

```
KontenB(3) := NEW KontoMitSperreTyp ("Wilram KG", 50_000.0, True);
```

Daß eine Verweisvariable nun Verweise auf Variable unterschiedlicher (aber verwandter) Typen aufnehmen kann, wird als *Polymorphismus* (Vielgestaltigkeit) bezeichnet.

Der Polymorphismus ermöglicht auch die gemeinsame Manipulation unterschiedlicher Variablen. Wenn z.B. von allen Konten in KontenB Kontoinhaber und Kontostand ausgedruckt werden sollen, kann dies folgendermaßen erfolgen:

```
FOR I IN 1..Anzahl_Konten
LOOP  Drucke ( KontenB(I)->.KontoInhaber ( ) , KontenB(I)->.Kontostand ( ) );
END LOOP;
```

7.2.3.13 Polymorphismus und Reimplementierung

Selbst wenn die angewendete Operation in den vorkommenden Typen unterschiedlich implementiert ist, kann eine gemeinsame Manipulation erfolgen; wenn z.B. auf alle Konten in KontenB eine Gutschrift von 1000.0 erfolgen soll:

```
FOR I IN 1..Anzahl_Konten
LOOP  KontenB(I)->.Gutschrift(1000.0);  END LOOP;
```

In diesem Beispiel treffen nun Polymorphismus und Reimplementierung zusammen. Da die Verweise in KontenB auf Objekte unterschiedlichen Typs verweisen können (z.B. KontoTyp oder KontoMitSperreTyp), stellt sich hier die Frage, welcher Rumpf der Prozedur Gutschrift in „KontenB(I)->.Gutschrift(1000.0)" aufgerufen wird, denn beide Typen haben ja jeweils einen eigenen Rumpf für diese Prozedur. Was passiert ist das, was man generell erwartet: verweist der Verweis KontenB(I) auf ein Objekt vom Typ KontoTyp, dann wird der in KontoTyp definierte Rumpf von Gutschrift ausgeführt und im Falle von KontoMitSperreTyp der in diesem Typ definierte Rumpf von Gutschrift. D.h. beim Prozeduraufruf über eine polymorphe Verweisvariable wird jeweils der Rumpf ausgeführt, der zu dem Typ des Objektes gehört, auf das die Verweisvariable gerade verweist. Diese Situation wird durch folgendes Programmfragment verdeutlicht:

```
VAR VerweisAufKonto : POINTER TO KontoTyp;
IF Bedingung
THEN  VerweisAufKonto := NEW KontoTyp ("XYZ AG", 0.0);
ELSE  VerweisAufKonto :=  NEW KontoMitSperreTyp ("ABC GmbH", 10_000.0, False);
END IF;

VerweisAufKonto->.Gutschrift(2500.0);
```

Wird der THEN-Zweig durchlaufen, dann wird KontoTyp.Gutschrift ausgeführt und im Falle des ELSE-Zweiges KontoMitSperreTyp.Gutschrift. Diese Vorschrift wird manchmal auch als „dynamisches Binden" bezeichnet. Trotz dieses Namens hat dieser Vorgang mit dem üblichen Binden von Programmteilen nichts zu tun. Aufgrund dieser mißverständlichen Ausdrucksweise, wurde schon häufig vermutet, daß der Aufruf von Prozeduren über polymorphe Verweisvariable besonders aufwendig sei. Dies ist aber nicht der Fall. Normalerweise wird die Vorschrift, daß jeweils die zum Typ der aktuellen Variablen gehörige Implementierung ausgeführt wird, durch einen indirekten Prozeduraufruf realisiert, der nur unwesentlich aufwendiger ist als ein direkter.

Es ist unmittelbar verständlich, daß nur die von KontoTyp exportierten Operationen in dieser Situation verwendet werden dürfen, denn die zusätzlichen, nur in KontoMitSperreTyp sind ja in den Variablen vom Typ KontoTyp nicht enthalten. D.h. das folgende ist fehlerhaft:

```
FOR I IN 1..Anzahl_Konten
LOOP  KontenB(I)->.Sperren( );  -- !!!!
END LOOP;
```

Umgekehrt garantiert die Vorschrift, daß im abgeleiteten Typ keine Komponenten des Basistyps weggelassen werden können, das sichere Zusammenspiel von Polymorphismus und Reimplementierung.

7.2.3.14 Dynamische Typabfrage

Falls es erforderlich ist, auch die nur in abgeleiteten Typen enthaltenen Prozeduren (oder anderen Komponenten) anzusprechen, dann muß festgestellt werden, von welchem aktuellen Typ die Variable ist, auf die eine polymorphe Verweisvariable verweist. Dafür werden in manchen objektorientierten Sprachen ebenfalls Sprachelemente angeboten. Im Rahmen dieser einführenden Darstellung kann darauf aber nicht eingegangen werden.

7.2.3.15 Benutzungsbeziehung

Neben der Ableitungsbeziehung, die zwischen Objekttypen bestehen kann, gibt es auf der Realisierungsebene noch eine zweite Beziehung zwischen Programmbausteinen: die Benutzungsbeziehung. Diese kann zwischen allen üblichen Programmbausteinen bestehen, also zwischen Objekttypen, Objekten / Modulen und Prozeduren: eine Prozedur kann einen Objekttyp benutzen oder ein Objekttyp kann ein Modul benutzen. Manchmal wird auch der Ausdruck „Baustein A importiert Baustein B" dafür benutzt.

Abb. I-7.2-13 zeigt eine Prozedur, die den Typ KontoTyp importiert.

```
PROCEDURE Hauptprogramm IS
        IMPORT KontoTyp;                    -- KontoTyp darf wie ein Typ verwendet werden
        CONST Anzahl_Konten = 1000;
        VAR Konten: ARRAY(1..Anzahl_Konten) OF KontoTyp;
BEGIN
        . . .
        Konten(5).Gutschrift(15_000.0);
        . . .
END Hauptprogramm;
```

Abb. I-7.2-13 Prozedur benutzt Objekttyp

Die Prozedur Hauptprogramm importiert den Typ KontoTyp und kann ihn dann intern wie einen Typ benutzen. Hier werden 1000 Konten vom Typ KontoTyp definiert und manipuliert. Die Verwendung eines importierten Objekttyps zur Definition entsprechender Variablen ist die typische Benutzung bei Import eines Typs. Die vom importierten Objekttyp selbst exportierten Größen werden dann in der Regel bei der Manipulation der entsprechenden Variablen benutzt; z.B. in Abb. I-7.2-13 in der Anweisung „Konten(5).Gutschrift(15_000.0);".

Bei Import eines Objektes / Moduls hingegen, werden die von diesem Objekt / Modul exportierten Größen im importierenden Baustein direkt benutzt. Dies ist in Abb. I-7.2-14 zu sehen, welche ein Pascal-Hauptprogramm zeigt, welches den in Abb. I-7.2-8 definierten Modul (unit) Konten importiert.

```
PROCEDURE Hauptprogramm2 IS
        IMPORT Konten;
        CONST Anzahl_Konten = 1000;
        VAR Kontenliste: ARRAY(1..Anzahl_Konten) OF Konten.KontoTyp;
                                                        Betrag: Konten.BetragsTyp;
BEGIN
        . . .
        Betrag := 15_000.0;
        Kontenliste(5).Gutschrift(Betrag);
        . . .
END Hauptprogramm;
```

Abb. I-7.2-14 Prozedur benutzt Modul

Ein Objekttyp kann auch andere Objekttypen importieren und benutzen. Abb. I-7.2-15 zeigt einen Objekttyp KontoFuehrungsTyp, der die weiter oben definierten Typen KontoTyp und KontoMitSperreTyp benutzt.

```
TYPE KontoFuehrungsTyp = OBJECT
        IMPORT KontoTyp, KontoMitSperreTyp;
        CONST Anzahl_Konten = 10_000;
        TYPE KontoNummerTyp = 1..Anzahl_Konten;
        PROCEDURE KontoEinrichten (Inhaber: IN KontoTyp.NamensTyp;
                                   Anfangsstand: IN KontoTyp.KontostandsTyp;
                                   MitSperre: IN Boolean
                                   KontoNr: OUT KontoNummerTyp);
        PROCEDURE KontoLoeschen (Inhaber: IN KontoTyp.NamensTyp;
                                   KontoNr: IN KontoNummerTyp);
        PROCEDURE Ueberweisen ( Von: IN KontoNummerTyp;
                                  Auf: IN KontoNummerTyp;
                                  Betrag: IN KontoTyp.BetragsTyp);
        . . .
INTERNAL
        VAR Konten: ARRAY(KontoNummerTyp) OF REF KontoTyp;
        . . .
END KontoFuehrungsTyp;
```

Abb. I-7.2-15 Objekttyp KontoFuehrungsTyp benutzt die Objekttypen KontoTyp und KontoMitSperreTyp

Betrachtet man die Abb. I-7.2-13, 14 und 15 genauer, dann stellt man fest, daß die Benutzungsbeziehung keinen direkten Einfluß auf den Aufbau des benutzenden Bausteins hat. Das Importieren eines Bausteins bedeutet, daß der importierende Baustein die in der Externschnittstelle des importierten Bausteins definierten Komponenten benutzen *kann*. In Abb. I-7.2-13 werden der importierte Typ KontoTyp und die in dessen Externschnittstelle definierte Prozedur Gutschrift verwendet. In Abb. I-7.2-15 werden der importierte Typ KontoTyp und die in dessen Externschnittstelle definierten Typen BetragsTyp und KontostandsTyp verwendet. Der Importeur ist aber nicht verpflichtet, etwas bestimmtes zu benutzen. Daher sagt auch die Importklausel selbst nichts näheres über die aktuelle Verwendung des importierten Bausteins aus.

Vergleicht man nun Ableitungs- und Benutzungsbeziehung, dann stellt man fest, daß die Ableitungsbeziehung eine relativ enge Kopplung der beteiligten Objekttypen zur Folge hat, während die Benutzungsbeziehung eine lose

Kopplung der beteiligten Bausteine mit sich bringt. Beide Beziehungen beschreiben Aspekte der statischen Programmstruktur. Dynamik und Funktionalität sind mehr implizit im Programmtext enthalten.

7.2.3.16 Merkmale der Objektorientierung auf Realisierungsebene

Objekt / Objekttyp :	Kombination von Daten und Operationen (Aggregierung)
Interne Komponenten :	Datenkapselung / Abstraktion
Ableitung / Vererbung :	Erweiterung / Modifikation (Reimplementierung) Schnittstellenmonotonie
Polymorphismus :	einheitliche Manipulation von Variablen verschiedenen Typs
Ableitungsbeziehung	enge Kopplung
Benutzungsbeziehung:	lose Kopplung

Abb. I-7.2-16 Merkmale der Objektorientierung auf Realisierungsebene

7.3 Programmentwicklung

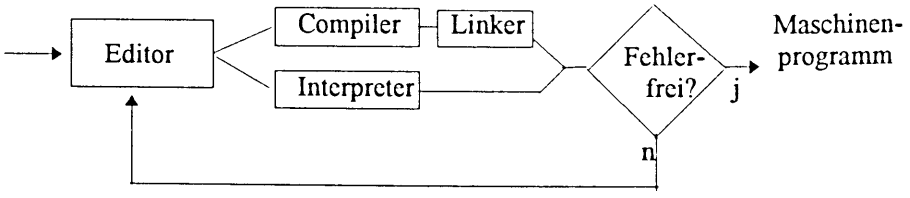
7.3.1 Zustandekommen lauffähiger Programme

Jedes Programm ist zunächst nur die Notation des Algorithmus in einer der Arbeitsweise des Menschen (nämlich der Umgangssprache) angepaßten Form. Diese Notation muß in den Befehlsvorrat der zugrunde gelegten Maschine übertragen werden, um ein lauffähiges Programm zu erhalten.

Moderne Programmiersprachen stellen die dazu erforderlichen Werkzeuge in einer integrierten Entwicklungsumgebung bereit:

Editor
Compiler
Linker
Interpreter
Debugger

Die erforderlichen Schritte sind



Programm erfassen Programm übersetzen Programm ausführen

Wirtschaftsinformatik

Anwendungsorientierte Einführung

Herausgegeben
von
Professor
Dr. Walter O. Riemann

unter Mitarbeit von

Prof. Dr. Manfred Goepel, Prof. Dr. Klaus Kruczynski,
Prof. Dr. Dr. Christian-Andreas Schumann,
Prof. Dr. Bernd Stöckert, Prof. Dr. Rolf Urban,
Prof. Dr. Lothar Wagner, Prof. Dr. Sabine Winkelmann,
Prof. Dr. Jürgen F. H. Winkler

2., völlig neu bearbeitete und erweiterte Auflage

R. Oldenbourg Verlag München Wien

Die Deutsche Bibliothek - CIP-Einheitsaufnahme

Wirtschaftsinformatik : anwendungsorientierte Einführung
hrsg. von Walter O. Riemann. Unter Mitarb. von Manfred
Goepel ... - 2., völlig neu bearb. und erw. Aufl. - München ;
Wien : Oldenbourg, 1996
1. Aufl. u.d.T.: Riemann, Walter O.: Betriebsinformatik
ISBN 3-486-23828-0
NE: Riemann, Walter O. [Hrsg.]; Goepel, Manfred

Gesamtherstellung: R. Oldenbourg Graphische Betriebe GmbH, München

ISBN 3-486-23828-0