Systemtechnische Sicht auf Objekte

Durch das Konzept des abstrakten Datentyps haben wir neben den globalen Variablen und den lokalen Variablen eine dritte Art von Variablen eingeführt. Während die Namen von globalen Variablen und lokalen Variablen zur Identifikation der zugehörigen Speicherzellen genügen (siehe Bild 53), gilt dies für die Namen der ADT-Variablen nicht, die zusammen mit den sogenannten Zugriffsprozeduren in der Kapsel des abstrakten Datentyps liegen. Die gesamte gekapselte Information wird ja mit gutem Grunde nicht als abstrakte Datenvariable sondern als abstrakter Datentyp bezeichnet. Ein Typ ist immer eine Beschreibung von möglichen Exemplaren, die gemäß dieser Beschreibung geschaffen werden können – man denke wieder an die Bauzeichnung einer Hundehütte, die der Herstellung vieler Exemplare von Hundehütten zugrunde gelegt werden kann. Damit es also Exemplare für die im abstrakten Datentyp enthaltenen Variablen geben kann, muss es im Programm explizite Exemplardeklarationen geben. Diese Exemplardeklarationen sehen syntaktisch genauso aus wie die Exemplardeklarationen für Variable der in der Programmiersprache eingebauten Typen. Wir haben dies schon im Falle des Stackbeispiels betrachtet.

Beispiele für Exemplardeklarationen:

```
INTEGER i,j,k;
STACK_OF_INTEGER s1,s2,s3;
```

Exemplare von abstrakten Datentypen werden üblicherweise als Objekte bezeichnet. Es ist üblich, mit dieser Art Objekt die Vorstellung zu verbinden, dass es ein Akteur sei, der ein Gedächtnis für seinen zeitveränderlichen Zustand hat und der bestimmte Arten von Aufträgen erledigen kann, die er von anderen Objekten erhält. Bei der Erledigung eines Auftrags kann das Objekt seinen Gedächtnisinhalt ändern, Aufträge an andere Objekte erteilen und am Ende dem Auftraggeber eine Ergebnisinformation liefern.

Die Vorstellung, dass jedes Objekt ein Akteur sei, ist nicht mit unserer Alltagsvorstellung vereinbar, wo wir zwischen aktiven und passiven Objekten unterscheiden. So stellen wir uns beispielsweise einen Stack nicht als einen Akteur vor, sondern als eine passive Datenstruktur. Aber da die Prozeduren zur Veränderung des Stacks im abstrakten Datentyp mit der Datenstruktur zusammen eine Einheit bilden, ist ein Stack als Exemplar eines abstrakten Datentyps mehr als die reine Datenstruktur. Man hat zwei Möglichkeiten, über dieses Exemplar zu reden je nachdem, welchen Interpreter man als den Ausführenden der Prozeduren zur Veränderung des Stacks ansieht. Wir wissen zwar, dass es in programmierten Systemen grundsätzlich die Hardware des Computers ist, welche die tatsächliche Interpretation der Prozeduren vornimmt. Wenn man aber seine Vorstellung an diese Realität bindet, schließt man anschaulichere Modellvorstellungen aus. Es ist sehr viel anschaulicher, sich die Vorgänge in programmierten Systemen als eine Kooperation mehrerer spezialisierter Akteure vorzustellen. Deshalb ist es durchaus sinnvoll, zwischen dem Stack als passivem Gedächtnisinhalt und dem Stack als einem zu Aktionen fähigem System, das ein Gedächtnis hat und Aufträge entgegennehmen kann, zu unterscheiden. Dabei mag es sinnvoll sein, sich nicht den Stack selbst als den Akteur vorzustellen, sondern sich in jedem Exemplar eines abstrakten Datentyps zusätzlich zur passiven Datenstruktur einen Verwalter vorzustellen, der für alle Aufträge, die diese Datenstruktur betreffen, zuständig ist.

Wenn man sich im Bereich der abstrakten Datentypen bzw. der zugehörigen Objekte sicher bewegen will, muss man eine klare Vorstellung davon haben, was mit bestimmten Bezeich-

nern gemeint ist. In der Sprache C haben wir den Begriff des Pointers oder Zeigers kennengelernt. In der graphischen Veranschaulichung ist ein Pointer ein Pfeil, der auf eine als Rechteck symbolisierte Speicherzelle zeigt (siehe Bild 59). Der Pointer ist Information zur Identifikation einer Speicherzelle, und diese Information, also der Pointer selbst, kann als „Wert“ in einer Speicherzelle liegen. Dann geht der Pfeil von der Speicherzelle aus, von der gesagt wird, sie enthalte den Pointer, und der Pfeil endet an der Speicherzelle, die durch den Pointer identifiziert wird.

Wenn die Zelle, in der der Pointer liegt, den Bezeichner `p` bekommen hat, wird durch die Symbolfolge `*p` die Zelle identifiziert, auf die der Pointer zeigt. Durch die Symbolfolge `&i`, worin `i` der Bezeichner für eine beliebige Speicherzelle sein soll, wird der Pointer, also in der Anschauung der Pfeil selbst identifiziert. `&i` identifiziert also nicht eine Speicherzelle, sondern nur einen möglichen Inhalt für eine Speicherzelle. So können wir durch die Zuweisungsanweisung `p:=&i` zum Ausdruck bringen, dass wir in die mit `p` bezeichnete Speicherzelle einen Pointer einspeichern wollen, der auf die mit `i` bezeichnete Zelle zeigt.

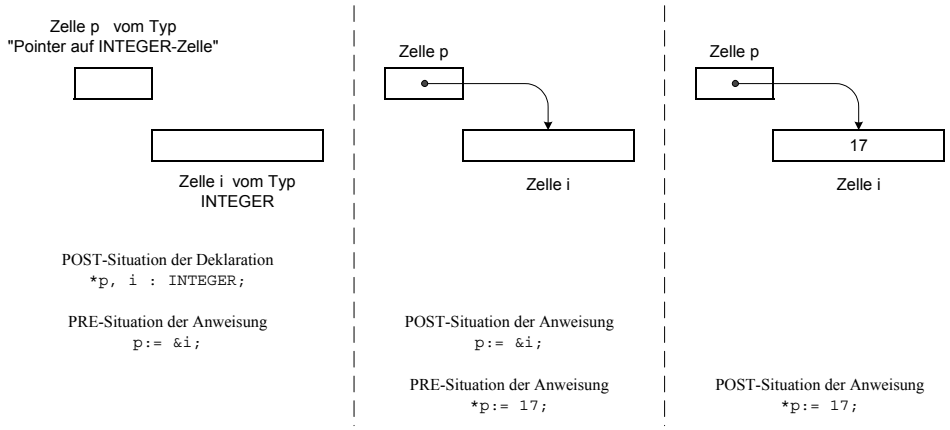

Bild 59 Veranschaulichung des Pointerbegriffs

So einfach wie im Bild 59 können die Verhältnisse nicht mehr sein, wenn wir Exemplare von abstrakten Datentypen betrachten. Während ein Exemplar vom Typ `INTEGER` eine Speicherzelle ganz bestimmter Kapazität ist, die im Speicher eindeutig lokalisiert werden kann, sind die Exemplare von abstrakten Datentypen meist Elemente von offenen Strukturen. Dies bedeutet, dass durch die Definition des abstrakten Datentyps im Allgemeinen keine Speicherstruktur fester Kapazität festgelegt wird, in die alle möglichen Werte hineinpassen. Man denke an den abstrakten Datentyp Stack in der Codierung als Pointerstruktur, wie sie in Bild 58 gezeigt ist.

Allen Exemplaren beliebiger offener Datenstrukturen ist nur noch das Merkmal gemeinsam, dass man sie jeweils durch genau einen Pointer eindeutig identifizieren kann. Wohin dieser identifizierende Pointer im jeweils konkreten Fall zeigen muss, hängt von der konkreten Implementierung des speziellen abstrakten Datentyps ab und kann nicht allgemein festgelegt sein. In Bild 60 sind die Verhältnisse dargestellt, wie sie im Falle der Stackimplementierung nach Bild 58 gelten.

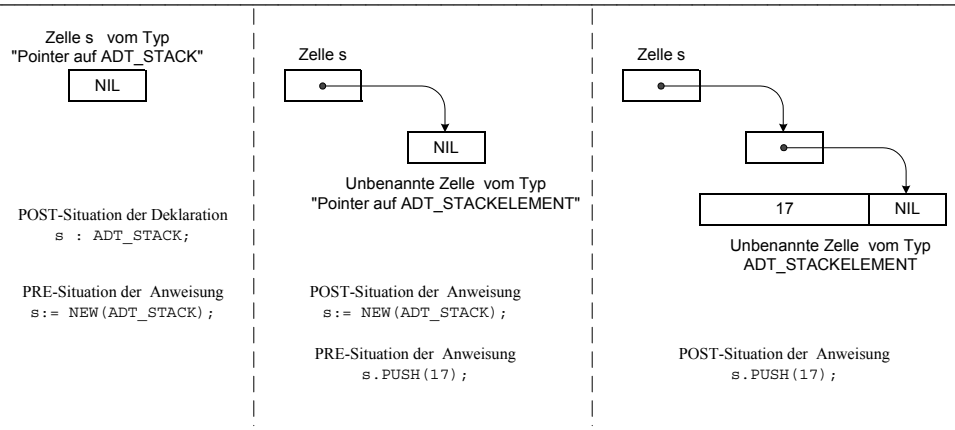

Bild 60 Veranschaulichung der Bedeutung von Ausdrücken bezüglich eines ADT

Wenn wir im Programm durch einen Ausdruck der Form

```
s : STACK;
```

zum Ausdruck bringen, dass wir ein Exemplar vom Typ Stack haben wollen, das wir durch den Bezeichner s identifizieren wollen, möchten wir mit diesem sprachlichen Ausdruck keine grundsätzlich andere Vorstellung verbinden müssen als bei dem Ausdruck

```
i : INTEGER;
```

Wir haben dabei die Vorstellung, dass durch diesen Ausdruck ein Behälter zur Verfügung gestellt wird, der durch den Bezeichner i identifiziert werden kann, aber dass dieser Behälter noch keinen definierten Inhalt hat. Im Unterschied zu den Exemplaren von `INTEGER` bestehen aber die Exemplare vom Typ `STACK` zwangsläufig immer aus zwei Teilen, nämlich der benannten Pointerzelle und der anfänglich leeren Speicherstruktur, in der später die Werte liegen werden. Bei den Exemplaren vom Typ `INTEGER` sind die benannten Speicherzellen selbst auch die Behälter für die Werte. In den Programmiersprachen, in denen abstrakte Datentypen formuliert werden können, äußert sich das Anlagen der Pointerzelle und das erstmalige Belegen der Pointerzelle mit einem Pointer i.a. in zwei getrennten programmiersprachlichen Ausdrücken, wie dies in Bild 60 gezeigt ist.

An dieser Stelle will ich noch einmal auf den Begriff des Containers zu sprechen kommen, der mit unterschiedlichen Exemplaren eines Typs belegt werden kann. Einen solchen Container können wir uns als Eimer vorstellen, auf den ein Etikett geklebt wurde, welches die Nutzung des Eimers einschränkt. Auf diesem Etikett könnte beispielsweise stehen „Nur für Wasser“ oder „Nur für trockene Pulver“. So wie es bei einem Eimer möglich ist, das aktuelle Etikett zu entfernen und durch ein anderes zu ersetzen, gibt es auch in manchen Programmiersprachen die Möglichkeit, eine bestehende Belegungsfestlegung für einen Container nachträglich zu ändern. Man spricht in diesem Fall von Casting.

Wenn ein Container für einen Datentyp angelegt wird, bei dem alle Exemplare den gleichen Speicherbedarf haben, kann der Container für diesen Speicherbedarf konkret eingerichtet werden. Wenn es sich dagegen um einen Container zu einem Datentyp mit offenem Speicherbedarf handelt, kann als Container nur eine Pointerzelle eingerichtet werden. Der Datentyp legt dann lediglich fest, auf welche Art von Zellen der Pointer zeigen darf.

Das Konzept des abstrakten Datentyps sollte man nicht nur dann verwenden, wenn man mit einer Programmiersprache programmiert, in der man abstrakte Datentypen explizit formulieren kann. Es ist durchaus zweckmäßig, das Konzept des abstrakten Datentyps auch zu ver-

wenden, wenn die Programmiersprache ohne Bezug zu abstrakten Datentypen konstruiert wurde. Man muss dann einfach mehr Disziplin wahren, denn die Speicherzellen, die im abstrakten Datentyp gekapselt sein sollen, müssen dann zwangsläufig als globale Variable eingeführt werden. Die Kapselung besteht in diesem Fall einfach im disziplinierten Verzicht auf direkte Zugriffe auf diese Variablen von außerhalb der ADT-Prozeduren. Ich erinnere an das Beispiel der Zugriffsprozedur `zufallszahl()`.

Objektorientierung

Das Verständnis des Begriffs des abstrakten Datentyps ist die Voraussetzung für die Einführung der objektorientierten Programmierung, mit der wir uns nun befassen wollen. In Bild 61 sind die Zusammenhänge veranschaulicht. Ein abstrakter Datentyp stellt eine Strukturbeschreibung dar, die auf alle Exemplare dieses Typs zutrifft. Diese Exemplare nennt man auch Objekte. Die Strukturbeschreibung besteht aus zwei Teilen:

- Im einen Teil werden die sog. Attributvariablen aufgezählt; das sind die „Gedächtniszellen“, die ein Objekt haben soll und deren Inhalt jeweils den aktuellen Zustand des Objekts darstellt. Der Typ einer Attributvariablen ist entweder ein elementarer Typ, der von der Programmiersprache angeboten wird, oder es ist ein Pointer auf ein Objekt, das während der Programmausführung erzeugt wird.
- Im anderen Teil werden die Prozeduren beschrieben, deren Ausführung vom Objekt verlangt werden kann. Diese Prozeduren werden Methoden genannt. Für diese Methoden sind die Attributvariablen „globale Variable“, d.h. eine Neubelegung einer solchen Variablen während der Ausführung einer Methode bleibt über die Ausführung dieser Methode hinaus erhalten.

Ein abstrakter Datentyp, der auch noch über Vererbungsbeziehungen und Polymorphiebeziehungen mit anderen abstrakten Datentypen in Beziehung stehen kann, wird Klasse genannt. Auch eine Klasse ist eine Typbeschreibung, zu der es Exemplare geben kann, die wieder Objekte genannt werden.

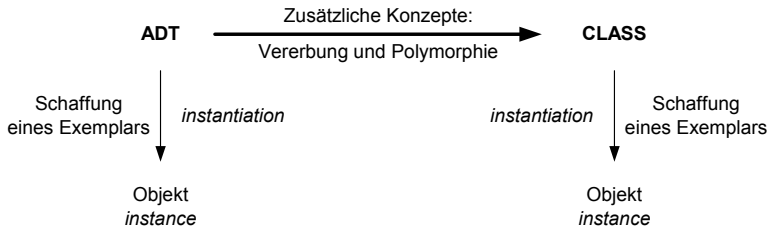

Bild 61 Zusammenhang zwischen den Begriffen „Abstrakter Datentyp“ und „Klasse“

Häufig wird gesagt, das Wesen der Objektorientierung bestehe in den Konzepten Kapselung, Vererbung und Polymorphie. Wir erkennen hier, dass die Kapselung schon ein Kennzeichen des abstrakten Datentyps ist und dass deshalb die Klasse aus dem abstrakten Datentyp durch Hinzufügen der beiden neuen Konzepte gewonnen wird.

Sprachverwirrung

An dieser Stelle will ich auf ein sprachliches Problem hinweisen, welches in der unseligen Gewohnheit der Deutschen besteht, englische Wörter, die eine gleichlautende deutsche Ent-

sprechung haben, ins Deutsche zu übernehmen und damit den ursprünglichen deutschen Sinn zu verdrängen. Das englische Wort für Exemplar ist *instance*, und in den Texten über Objektorientierung hat man nun das deutsche Wort Instanz einfach für alle Fälle genommen, wo es im Deutschen Exemplar heißen müsste. Man spricht auch bei der Schaffung eines Exemplars zu einem Typ vom Instanziieren. Eigentlich bedeutet Instanz im Deutschen eine Institution, die für eine bestimmte Aufgabe zuständig ist. Insbesondere im Gerichtswesen spricht man von den verschiedenen Instanzen.

Andere Wörter, bei denen eine entsprechende Fehlverwendung eingetreten ist, sind die Wörter adressieren, unterstützen und kontrollieren. Im Deutschen war es früher völlig sinnlos zu sagen: “Wir haben ein bestimmtes Problem noch nicht adressiert.“ Im Deutschen adressiert man eigentlich nur Postsendungen. Während es im Englischen völlig korrekt ist zu sagen: *We did not yet address this problem*. müsste man im Deutschen korrekterweise sagen: „Mit diesem Problem haben wir uns noch nicht befasst.“ Das englische Wort *control* entspricht im Deutschen dem Wort „steuern“ und nicht dem Wort „kontrollieren“. Kontrollieren im Deutschen bedeutet das Überprüfen. So kann beispielsweise der Vorgesetzte kontrollieren, ob sein Nachtwächter um Mitternacht wach ist oder schläft. Im Deutschen dagegen ist es nicht korrekt zu sagen, man habe die Kontrolle über sein Fahrzeug verloren. Man müsste vielmehr korrekterweise sagen, man habe die Beherrschung über das Fahrzeug verloren.
Im dritten Beispiel geht es um das Wort „unterstützen“. So wird beispielsweise gesagt, diese oder jene höhere Programmiersprache unterstütze das Konzept der abstrakten Datentypen, und eine andere unterstütze dieses Konzept nicht. Was man eigentlich meint ist der Sachverhalt, dass in der einen Sprache abstrakte Datentypen formuliert werden können und in der anderen Sprache nicht. Man gewöhnt sich sehr leicht diese schlampige Sprache an, und da ich selbst feststelle, dass ich manchmal einen solchen Unsinn rede, kann ich es Ihnen auch nicht übel nehmen, wenn Sie so reden. Ich muss es Ihnen aber übel nehmen, wenn Sie so schreiben. Denn beim Schreiben muss man sich mehr Mühe geben als beim Sprechen.

Vererbung

Nun werden wir den Begriff der Vererbung betrachten, der von den abstrakten Datentypen zu den Klassen der objektorientierten Programmierung führt. Den Begriff der Polymorphie, der ebenfalls in diesen Zusammenhang gehört, können wir erst behandeln, nachdem der Begriff Vererbung vollständig eingeführt ist, denn Polymorphie hängt sehr streng mit der Vererbung zusammen.

Der Vererbungsbegriff ist mehrdeutig, d.h. er kommt in zwei sehr unterschiedlichen Kontexten vor. Der eine Kontext ist die Juristerei. Dort wird die Weitergabe von Eigentum im Falle eines Todes festgelegt. Jemand stirbt, und andere erben. Dieser Kontext war es nicht, der den Vererbungsbegriff in der Objektorientierung prägte.
Vielmehr war es der Kontext der Biologie, wo es eine Vererbungslehre gibt, die die Gesetzmäßigkeiten beschreibt bei der Weitergabe von Eigenschaften der Eltern auf ihre Nachkommen. Hier behält der Vererbende das, was er weitergibt, denn die Weitergabe erfolgt durch Kopie. So verliert beispielsweise ein Mann nicht seine Veranlagung, eine Glatze zu bekommen, wenn er diese Veranlagung an seinen Sohn vererbt.

Bei dem Vererbungskonzept in der Programmierung geht es immer um die Ausnutzung von Ähnlichkeitsbeziehungen zwischen Typen. Von der Möglichkeit, eine Vererbungsbeziehung zu formulieren, kann man immer dann Gebrauch machen, wenn man schon einige Typen de-

finiert hat und nun neue Typen hinzufügen muss. Bei den neuen Typen könnte es zweckmäßig sein, auf die bereits definierten Typen Bezug zu nehmen. Dadurch kann sich der Aufwand zur Definition der neuen Typen reduzieren, weil man Aussagen, die bei den neuen Typen genauso heißen wie bei den bisherigen Typen, nicht mehr neu formulieren muss.

Eine Klasse in der Objektorientierung ist - genauso wie ein abstrakter Datentyp - bestimmt durch die Menge der gekapselten Variablen und der Zugriffsprozeduren, die hier Attribute und Methoden genannt werden. Eine besonders einfache Form der Vererbungsbeziehung besteht darin, dass die Attributmenge und die Methodenmenge des Erbenden Erweiterungen dieser Mengen des Vererbenden sind. Diese Beziehung ist links im Bild 62 veranschaulicht.

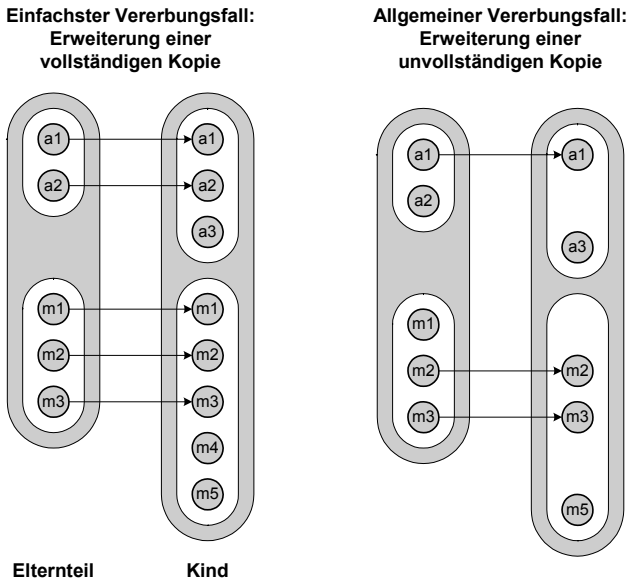

Bild 62 Vererbungsvarianten

Wenn man an die Vererbung zwischen Eltern und Kindern denkt, sind aber auch kompliziertere Vererbungsbeziehungen denkbar: Manche Eigenschaften der Eltern sind bei den Kindern auch vorhanden, andere Eigenschaften der Eltern fehlen bei den Kindern, und manche Eigenschaften sind bei den Kindern vorhanden, aber bei den Eltern nicht. Eine solche Vererbungsbeziehung ist im Bild 62 rechts veranschaulicht.

Von besonderer Bedeutung ist in der Programmierung nur die einfache Vererbungsbeziehung, bei der alle Eigenschaften von den Eltern auf die Nachkommen übertragen werden. Bezüglich der Methoden gibt es allerdings die Möglichkeit, dass nur die Signaturen unverändert vererbt werden; der innere Prozedurtext darf beim Kind anders sein als bei den Eltern.

Der Begriff „Klasse“

Die Verwendung der Bezeichnung Klasse anstelle des Wortes Typ führt oft zu einer gewissen Verwirrung. Deshalb will ich hierzu einige Überlegungen vorstellen. Klassifikation ist ein uralter Begriff aus der Philosophie, der nicht erst im Zusammenhang mit Computern und Software erfunden wurde. Klassifikation dient dem Menschen zur Strukturierung seiner Welt, die mit riesigen Mengen von Erscheinungen gefüllt ist. So hat der Mensch beispielsweise die Menge der Tiere eingeteilt, d.h. er hat die Exemplare klassifiziert, indem er von bestimmten

konkreten Eigenschaften des einzelnen Tieres abstrahiert hat. So hat er beispielsweise die Klasse der Wirbeltiere geschaffen. In der Klasse der Wirbeltiere gibt es etliche disjunkte Teilklassen, beispielsweise die Säugetiere und die Vögel. In der Klasse der Säugetiere gibt es wieder Unterklassen, beispielsweise die Klasse der Katzenartigen und die Klasse der Primaten. Die Klassen und ihre Teilklassen bilden einen Klassifikationsbaum. Die Exemplare der Klassen gehören nur zu den Blattklassen. Wenn es also in diesem Baum einen Knoten für die Klasse der katzenartigen Tiere gibt, über dem der Knoten der Klasse der Säugetiere sitzt, wird eine ganz konkrete Hauskatze im Graphen an der Klasse der Katzenartigen hängen und nicht an der Klasse der Säugetiere. Dass die konkrete Katze auch zur Klasse der Säugetiere gehört, wird in der graphischen Darstellung in Bild 63 nicht durch eine direkte Linie zwischen dem Katzenknoten und dem Säugetierknoten zum Ausdruck gebracht, sondern durch den Pfad, der vom Katzenknoten über den Knoten der Katzenartigen zum Säugetierknoten führt.

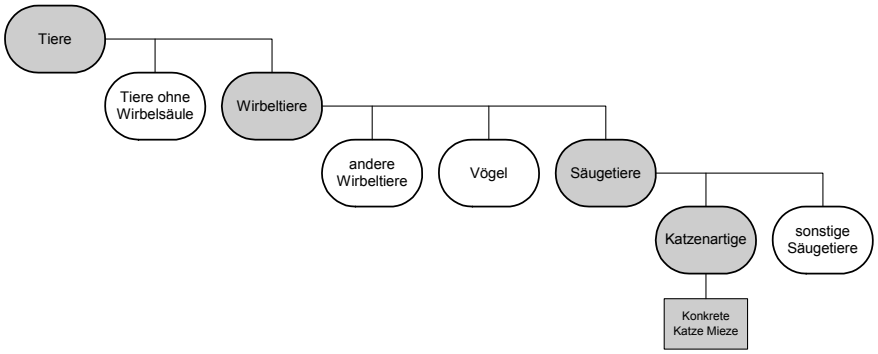

Bild 63 Beispiel eines Klassifikationsbaums

Wir können also feststellen, dass es in einem Klassifikationsbaum zu Klassenknoten, die selbst wieder in Unterklassen aufgeteilt sind, keine direkten Exemplare, sondern nur indirekte Exemplare gibt. Man könnte dies auch dadurch ausdrücken, dass man sagt, in einem Klassenbaum, dessen Struktur durch die Klassenabstraktion definiert ist, sind nur die Blattklassen konkrete Klassen, während alle anderen darüber liegenden Klassen abstrakte Klassen sind. Wir nennen eine Klasse abstrakt, wenn es zu ihr keine direkten Exemplare gibt, und wir nennen sie konkret, wenn es zu ihr direkte Exemplare gibt.

In der objektorientierten Programmierung gibt es nun durchaus den Fall, dass eine Klasse, die im Klassenbaum kein Blatt ist, trotzdem eine konkrete Klasse sein kann, zu der es direkte Exemplare gibt. Daraus muss geschlossen werden, dass die Beziehung zwischen den Klassen, welche die Struktur des Baumes definiert, keine reine Abstraktionsbeziehung mehr sein kann. Dies ist nicht verwunderlich, denn im Falle der objektorientierten Programmierung ist der Klassenbaum ein Vererbungsbaum und nicht ein Abstraktionsbaum. In einem Vererbungsbaum kann jede Klasse konkret sein, d.h. es kann zu jeder Klasse unabhängig von ihrer Lage im Baum direkte Exemplare geben. Während im Klassifikationsbaum, wo die Klassen in einer Abstraktionsbeziehung zueinander stehen, die abstrakte Klasse eine Selbstverständlichkeit darstellt, ist im Falle des Vererbungsbaums die konkrete Klasse die Selbstverständlichkeit. Es kann aber auch im Vererbungsbaum abstrakte Klassen geben. Dies ist eine Konsequenz der Möglichkeit, eine Methodendefinition in zwei Teile zu zerlegen, die in zwei getrennten Klassen definiert werden. Die Signatur kann beim Elternteil definiert sein, aber der Rest der Prozedur kann möglicherweise erst beim Kind stehen. Dann kann es vom Elternteil keine direkten Exemplare geben, denn eine Methode im Exemplar muss zwangsläufig vollständig definiert sein.

Solange man den auf der Klassenabstraktion beruhenden Klassenbaum und den Vererbungsbaum begrifflich nicht durcheinander bringt, kann man über die Sachverhalte sehr verständlich reden. Zur Veranschaulichung des Unterschieds zwischen diesen beiden baumdefinierenden Beziehungen sind in Bild 64 ein Klassifikationsbaum und ein Vererbungsbaum einander gegenüber gestellt. Die betrachteten Exemplare sind Tische, die in die drei Klassen dreibeinige Tische, vierbeinige Tische und sonstige Tische eingeteilt wurden. Der linke Baum enthält die abstrakte Klasse Tisch, die sich als Abstraktion der drei konkreten Tischklassen ergibt. Rechts im Bild kommt gar keine abstrakte Klasse vor. In dieser Vererbungsbeziehung gibt die Klasse der dreibeinigen Tische die Eigenschaft weiter, dass mindestens 3 Beine vorhanden sein müssen. Die Klasse der vierbeinigen Tische unterscheidet sich also von der Klasse der dreibeinigen Tische durch eine Erweiterung. Entsprechendes gilt für das Verhältnis zwischen der Klasse der vierbeinigen und der sonstigen Tische.

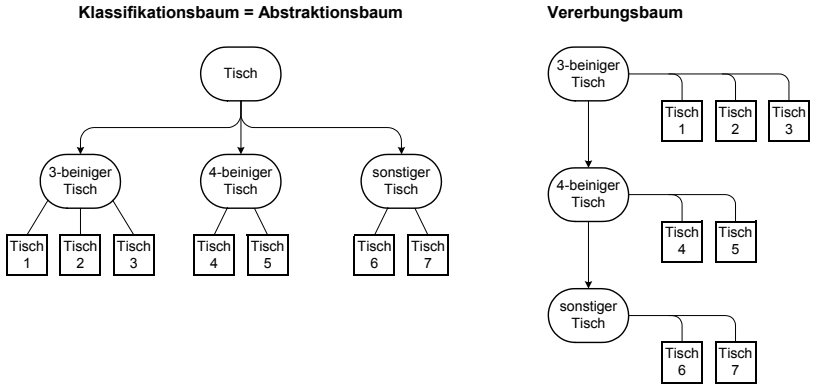

Bild 64 Unterscheidung zwischen Abstraktion und Vererbung

Wenn wir die Vererbung ohne Bezug zur Polymorphie betrachten, bringt sie zwei Vorteile. Zum einen hilft sie, unser Denken über die behandelten Objekte zu strukturieren, und zum anderen dient sie der Beschreibungsökonomie, d.h. der Reduktion des Schreibaufwandes. Wenn eine neue Klasse definiert werden soll und man sich dabei auf bereits definierte Klassen beziehen kann, braucht man gewisse Dinge in der neuen Klasse nicht mehr zu schreiben, sondern kann einfach auf die bisherigen Klassen verweisen. Solche Verweise sind nicht erst mit der objektorientierten Programmierung in die Welt gekommen: man kennt das ja schon aus vielen Schriften, dass irgendwo steht „siehe Kapitel 2“. Man erspart sich dadurch die Notwendigkeit, das was in Kapitel 2 schon geschrieben wurde, nun noch einmal hinschreiben zu müssen.

Mehrfachvererbung und Interfaces

Bei der Objektorientierung hat sich neben dem Begriff der Klasse der Begriff des Interfaces eingebürgert. Ein Interface kann als abstrakte Klasse betrachtet werden, in der keine Attribute und keine Methodenimplementierungen enthalten sind. Ein Interface ist also eine reine Sammlung von Signaturen. Während bei Klassen die sogenannte Mehrfachvererbung Probleme macht, gibt es diese Probleme der Mehrfachvererbung im Interfacebereich nicht. In Bild 65 ist gezeigt, weshalb die Mehrfachvererbung bei Klassen problematisch ist.

Das Problem kommt daher, dass eine Methode sowohl eine Signatur als auch eine Implementierung braucht. Zu ein- und derselben Signatur kann es mehrere unterschiedliche Implemen-

tierungen geben, die der Signatur auch eine andere Semantik zuordnen. Bei der mehrfacherbenden Klasse ist dann die Entscheidung offen, welche Implementierung denn nun gelten soll.

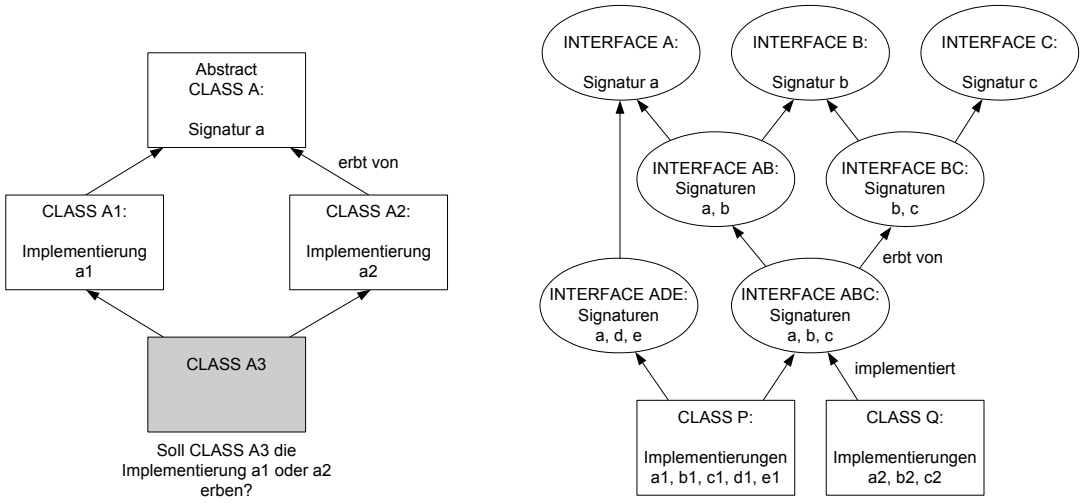

Bild 65 Klassen, Interfaces und Mehrfachvererbung

Da es bei Interfaces keine Implementierungen gibt, kann dieses Problem gar nicht entstehen.Deshalb führt man Sprachen ein, bei denen die Mehrfachvererbung auf die Interfaces beschränkt ist. Die Vererbung ist eine Beziehung, die sowohl zwischen zwei Interfaces als auch zwischen zwei Klassen bestehen kann. Man braucht aber auch noch eine Beziehung zwischen Interfaces und Klassen. Dies ist die Implementierungsbeziehung, d.h. man sagt, eine Klasse implementiert ein Interface, wenn diese Klasse Implementierungen zu allen Signaturen enthält oder geerbt hat, die in dem Interface vorkommen.

Polymorphie

Polymorphie bedeutet Vielgestaltigkeit. Man kann diesen Begriff in der objekt-orientierten Programmierung nur verstehen, wenn man den Unterschied zwischen kleinen Klassen und großen Klassen im Vererbungsbaum vor Augen hat. In Bild 66 ist ein Vererbungsbaum dargestellt.

Es wurde angenommen, dass alle 6 Klassen A bis F konkrete Klassen sind, d.h. dass es zu allen diesen Klassen Exemplare geben kann. Sämtliche Exemplare in Bild 66 sind Exemplare der großen Klasse A. Zwei dieser Exemplare der großen Klasse A sind auch Exemplare der kleinen Klasse A. Die große Klasse A wird gebildet durch die Vereinigung aller Klassen in dem Baum, dessen Wurzel mit A bezeichnet ist. Da ein Baum nur eine einzige Wurzel hat, kann jeder Baum durch Identifikation seiner Wurzel identifiziert werden. Dadurch ergibt sich eine Mehrdeutigkeit des Bezeichners A, denn entweder meint man damit nur den Wurzelknoten, also die kleine Klasse A, oder man meint damit den ganzen Baum, also die große Klasse A.

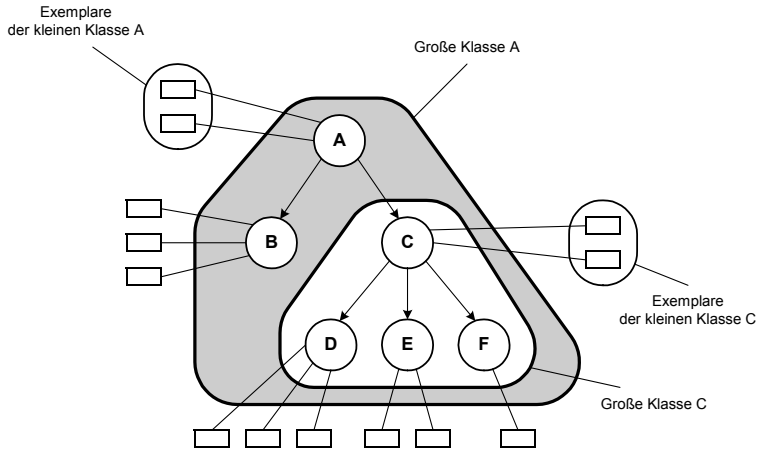

Bild 66 Zur Unterscheidung von kleinen und großen Klassen im Vererbungsbaum

Man denke hier an die anschauliche Analogie zwischen einer Baumwurzel und dem Passbild einer Person. So wie die Identifikation einer Baumwurzel ausreicht, den ganzen Baum zu identifizieren, reicht das Passbild aus, die ganze Person zu identifizieren, obwohl es unmittelbar eigentlich nur den Kopf einer Person identifiziert. Es wäre unsinnig zu sagen, das Passbild identifiziere eine Person, die keinen Rumpf hat.

Im Programmtext sind nur den kleinen Klassen eindeutige Textabschnitte zugeordnet, nämlich die Klassendefinitionen. Wenn man also Aussagen über die große Klasse ins Programm schreiben will, muss man dies trotzdem beim Text der kleinen Klasse unterbringen. Wenn der Klassenbezeichner im Programmtext verwendet wird, sieht man dem Bezeichner selbst zwar nicht an, ob er zur Identifikation der kleinen Klasse oder der großen Klasse benutzt wird, aber diese Entscheidung ist doch eindeutig aus dem Kontext zu entnehmen. Durch die Programmzeile `a1 : CLASS_A;`
wird verlangt, dass eine Speicherzelle eingerichtet wird, in die ein Pointer gebracht werden kann, der auf ein Objekt der Klasse A zeigt. Hier ist die große Klasse A gemeint, d.h. der Pointer könnte auf jedes Exemplar aus Bild 66 zeigen. Wenn nach dieser Pointerdeklaration durch die Programmzeile `a1:= NEW(CLASS_A);`
die Pointerzelle mit einem konkreten Pointer belegt wird, zeigt dieser Pointer auf ein Exemplar der kleinen Klasse A, weil der Klassenbezeichner A im Kontext des `NEW`-Operators verwendet wurde. Es wäre auch die Zuweisung `a1:= NEW(CLASS_F);`
zulässig, bei der in die Pointerzelle `a1` ein Pointer auf ein Exemplar der kleinen Klasse F gebracht wird. Das Exemplar der kleinen Klasse F ist auch ein Exemplar der großen Klasse A, und deshalb kann die Pointerzelle entsprechend belegt werden, denn sie wurde ja als Behälter für Pointer auf Exemplare der großen Klasse A geschaffen.

Es gibt Programmiersprachen, bei denen kann die Kapselung detaillierter formuliert werden als nur unter Verwendung der Vererbungsbeziehungen. Man findet insbesondere die folgenden Nutzungsprädikate:

private	Zugriff nur in Methoden der kleinen Klasse
protected	Zugriff nur in Methoden der großen Klasse
public	Zugriff in allen Methoden

Der Begriff der Polymorphie bringt zum Ausdruck, dass Pointerbehälter geschaffen werden können für große Klassen, wobei der jeweils aktuelle Pointer in diesem Behälter stets auf ein Exemplar einer kleinen Klasse zeigt, die ein Knoten im Vererbungsbaum ist, dessen Wurzel die große Klasse identifiziert.

Polymorphie bringt den Vorteil des sogenannten „CASE-less programming“. Dieser Begriff wurde geschaffen in Analogie zum Begriff „GOTO-less programming“, mit dem das strukturierte Programmieren gemeint ist und der auf den Titel von Dijkstras Aufsatz verweist „The GOTO-Statement Considered Harmful“. Mit CASE-less programming ist die Möglichkeit gemeint, die explizite Formulierung bestimmter Fallunterscheidungen im Programmablauf zu vermeiden, indem man diese Fallunterscheidung in den Vererbungsbaum transferiert. In Bild 67 sind diese beiden Arten der Fallunterscheidung veranschaulicht: Links im Bild sieht man eine Unterscheidung von 3 Fällen in einem Petrinetz, wobei die Auswahl des jeweiligen Falles durch die Feststellung bestimmt wird, von welchem Typ ein bestimmtes Objekt ist, das hier Gegenstand der Verarbeitung ist. Rechts im Bild ist gezeigt, dass im Petrinetz auf die Fallunterscheidung verzichtet werden kann, wenn im Vererbungsbaum die drei Fälle explizit unterschieden werden. In der Transition des Petrinetzes steht nun kein Algorithmus mehr, sondern nur noch der Aufruf eines Algorithmus, d.h. der Aufruf einer Methode, und dazu wird im Ablauf nur die Signatur gebraucht. Diese Signatur kann für alle drei unterschiedlichen Algorithmen die gleiche sein, und deshalb wird sie bereits in der Wurzel des Vererbungsbaums eingeführt.

Die unterschiedlichen Algorithmen, die zu dieser Signatur gehören, stehen in den unterschiedlichen Verzweigungsfällen des Vererbungsbaumes.

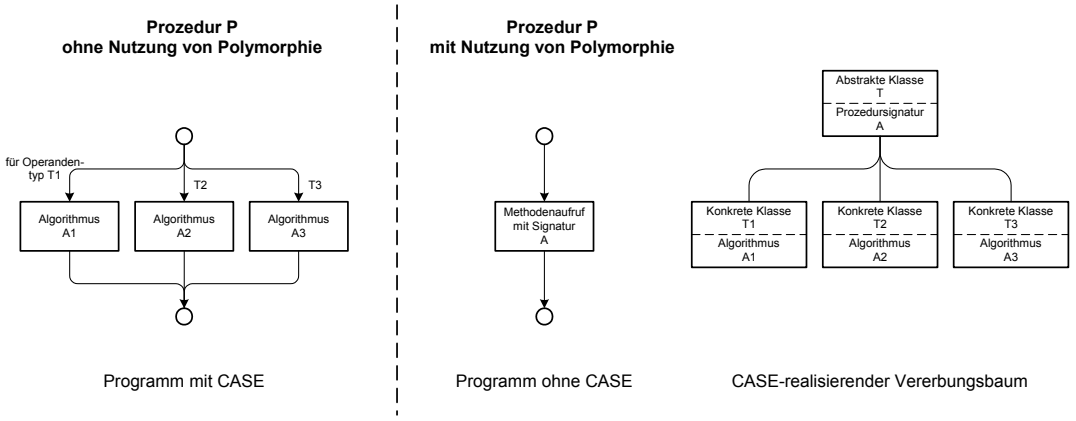

Bild 67 Zur Veranschaulichung von „CASE-less programming“ durch Polymorphie