Objektorientierte Spezifikation: Konzepte und eine Notation

W. Bartsch
E. Denert

Fakultät für Informatik
Technische Universität München
Arcisstraße 21
80333 München

und

sd&m gmbh
Thomas-Dehler-Straße 18
81737 München

E-Mail: bartsch@informatik.tu-muenchen.de

1 Zur Bedeutung der Spezifikation

Die Bedeutung der Spezifikation eines Informationssystems kann gar nicht hoch genug eingeschätzt werden. Sie ist die Grundlage der weiteren Softwareentwicklung und des Projekterfolgs. Wie kommt man zu einer guten Spezifikation? Der Titel dieses Beitrags suggeriert, man müsse sie nur objektorientiert machen und ein gutes Ergebnis sei damit schon gesichert. Weit gefehlt — entscheidend ist vielmehr, daß die Designer das Anwendungsgebiet sehr gut verstehen, daß sie in der Lage sind, ein zukunftweisendes Fachkonzept und daraus eine gut strukturierte Anwendungsarchitektur zu entwickeln sowie schließlich eine detaillierte und präzise Spezifikation zu verfassen. Dazu bedarf es einer intensiven Zusammenarbeit mit den Anwendern. Die objektorientierte Methodik[1] hilft (viel besser als andere Methoden), die gewonnenen Erkenntnisse gut strukturiert auszuformulieren, aber die fachliche Durchdringung und kreative Gestaltung der Anwendung ist die entscheidende Denkarbeit der Designer.

Die Spezifikation beschreibt das Was eines Informationssystems, definiert also aus Außen- bzw. Anwendersicht die Anforderungen, denen die zu entwickelnde Software zu genügen hat. (Dafür benutzen wir das Wort Analyse nicht gern, das sich — aus dem

[1] Zur Diskussion der Begriffe "Methodik" und "Methode" siehe [Löhr93]

Amerikanischen kommend — auch bei uns eingebürgert hat; denn eine Spezifikation legt fest, was sein soll, eine Analyse liefert aber nur die Erkenntnisse über das, was ist.).

Eine Spezifikation hat die Anforderungen bezüglich dreier Bereiche zu definieren: Daten, Funktionen und Schnittstellen (zu Benutzern und anderen Systemen). Das führt dann zu der "klassischen" Dreiteilung einer Spezifikation wie in Abb. 1a. Das Grundprinzip der objektorientierten Methodik ist dagegen die innige Verbindung von Daten und Funktionen, demzufolge sich eine Trennung von Daten- und Funktionenmodell eigentlich verbietet und das eine Sicht wie in Abb. 1b nahelegt. An ihr orientieren wir uns im folgenden.

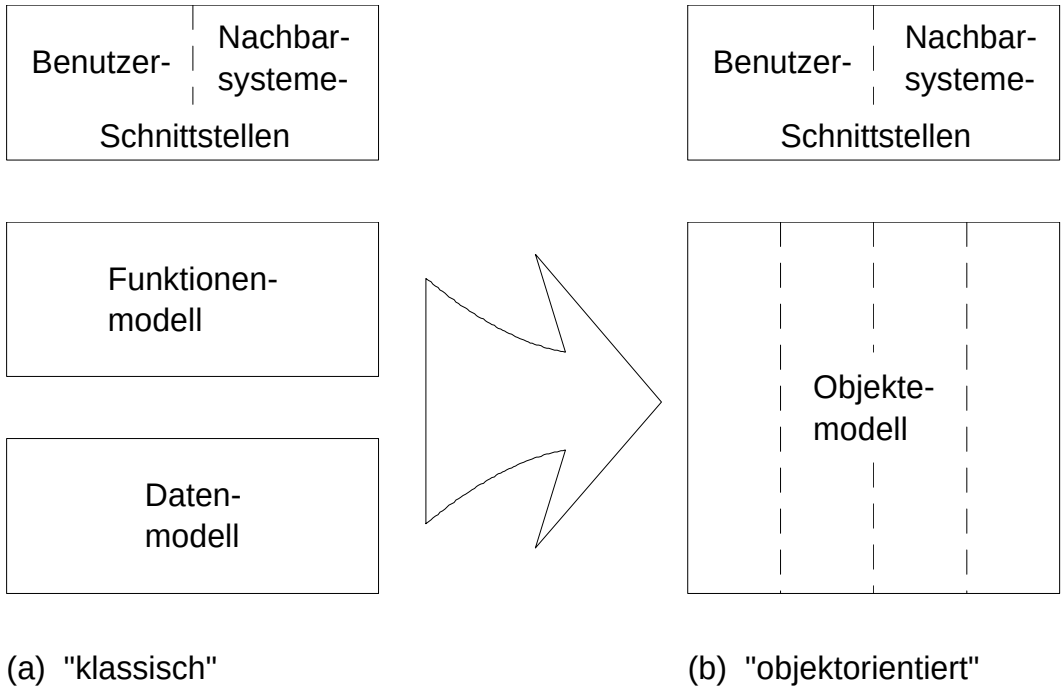

Abb. 1 Die Teile einer Spezifikation

Ein wesentlicher Inhalt einer objektorientierten Spezifikation ist also die Beschreibung der Objekte und ihres Zusammenspiels. Wie aber findet man sie? Man nähert sich der Anwendung am besten von zwei Seiten:

- Die Anforderungen resultieren letztlich aus den *Geschäftsvorfällen*, deren Bearbeitung das zu entwickelnde System unterstützen soll. Für jeden Geschäftsvorfall ist (mindestens) eine Funktion eines Objekts zu spezifizieren. Es ist dann eine der Aufgaben der Benutzerschnittstelle zu erkennen, welchen Geschäftsvorfall der Anwender bearbeiten will, und die dafür nötige Objektfunktion ausführen zu lassen (aufzurufen).

- Ein eigenständiges *Datenmodell*, etwa in Entity/Relationship-Darstellung, hat in einer echt objektorientierten Spezifikation eigentlich keine Existenzberechtigung

mehr; dennoch lohnt es sich, die Spezifikationsarbeiten damit zu beginnen. Die Entitäten, die man dabei findet, bilden sehr oft Kristallisationskeime für Objekte, und das schließlich entstehende Objektebild hat viel Ähnlichkeit mit dem E/R-Diagramm. Es ist allerdings zu einfach zu glauben, man müsse an die Entitäten nur ein paar Funktionen kleben — und fertig sei das Objektemodell.

Nehmen wir an, durch Betrachtung des E/R-Datenmodells, durch die Analyse der Geschäftsvorfälle und durch andere Überlegungen seien Objekte als Einheiten von Daten und Funktion — genauer: Objekttypen (= Klassen) — gefunden und die Beziehungen zwischen ihnen etabliert worden. In einem *E/R*- sowie einem *Objekte-Bild* wird eine Übersicht der Entitäten bzw. Objekte mit ihren Beziehungen gegeben. Dann gilt es, jeden einzelnen *Objekttyp* präzise zu *spezifizieren*. Dazu verfassen wir je Objekttyp *ein Dokument* in einer Art und Weise, die wir im folgenden erläutern und motivieren wollen.

Auf eine ausführliche Diskussion der Vor- und Nachteile einer objektorientierten Methode verzichten wir hier und verweisen auf die Literatur ([Endres92], [Rumbaugh91], [Coad91], [Meyer88]).

2 Objektorientierte Konzepte in der Spezifikation

Was ist ein Objekt?

"Im allgemeinen Sprachgebrauch meint man damit etwas fest Umrissenes mit einer klaren Grenze, z.B. einen Baum, ein Buch, einen Planeten." [Denert91].

Die Objekte hingegen, denen wir bei der Spezifikation eines Informationssystems begegnen, sind *Modelle* der realen Objekte seiner Anwendung. Dabei enthält dieses Modell nur die für die Anwendung relevanten Aspekte des Objektes. Es folgen einige Ausführungen zu den Konzepten, die ein Objekt charakterisieren und für seine Spezifikation wichtig sind.

Eigenschaften

Wir beschreiben ein Objekt (genauer: dessen Modell), indem wir die *Merkmale (attributes)* und *Fähigkeiten (methods)* des Objekts beschreiben. Die Merkmale sind dabei die Größen, die man dem Objekt zuordnen kann, also zum Beispiel Gewicht, Umfang, Alter, Farbe o.ä.; Fähigkeiten sind dann die Handlungen und Aktionen, die ein Objekt ausführen kann oder die man am Objekt ausführen kann. Die Merkmale entsprechen also, technisch gesprochen, den Attributen, die Fähigkeiten den Funktionen oder "Methoden" der Objekte. Als Oberbegriff von Merkmal und Fähigkeit verwenden wir das Wort *Eigenschaft (feature)*.

Fähigkeiten

Objekte haben Fähigkeiten. Mit ihren Fähigkeiten verändern sie, und nur sie die Werte ihrer Merkmale. Wir nennen dies die *Vorstellung vom aktiven Objekt* oder die *animistische*[2] *Auffassung*, denn wir verbinden mit Objekten, die man für scheinbar "tot" hält oder die gar nur ein abstrakter Begriff sind, die Vorstellung, daß sie aktiv ihre Fähigkeiten anbieten und wissen, wie sie handeln müssen, wenn eine ihrer Fähigkeiten beansprucht wird. Objekte wie *Wertpapiere* und sogar *Konten* können also mittels ihrer Fähigkeiten aktiv auf sich und ihre Umwelt einwirken.

Fähigkeiten haben (wie Prozeduren in Programmiersprachen) Ein- und Ausgaben. Die Eingaben werden verwandt, um den Ablauf einer Fähigkeit oder Ergebnisse zu beeinflussen. Ausgaben sind die Ergebnisse, die die Fähigkeit aus den Merkmalswerten ermittelt. Darüber hinaus hinterläßt das Anwenden einer Fähigkeit Spuren (nämlich Wertänderungen) in den Merkmalen des ausführenden Objekts.

Die Fähigkeiten eines Objekts können von anderen Objekten benutzt werden, um gemeinsam Aufgaben zu erledigen. Ein Objekt kann also die Fähigkeiten anderer Objekte nutzen und in eigenen Fähigkeiten verwenden.

Merkmal oder Fähigkeit?

Der Übergang von Merkmalen, also "gespeicherten Informationen", zu Fähigkeiten ist fließend. So kann man beispielsweise den *Rohertrag* aus *Verkaufspreis* und *Einkaufspreis* berechnen. Aus fachlicher Sicht sind aber alle drei Eigenschaften dieses Objekts (z.B. des Objekts *Ware*) Merkmale, und keine Fähigkeiten. Eine Fähigkeit *RohertragBerechnen()*[3] wäre ziemlich trivial und hinterließe vor allem keine Spuren in den Merkmalen der *Ware*, d.h. in einer Programmiersprache wie Pascal würde man sie als "function" (ohne Seiteneffekt) und nicht als "procedure" mit Seiteneffekt realisieren. Darüber hinaus würde diese Art der Spezifikation eine nicht erwünschte Auszeichnung des *Rohertrag*s nach sich ziehen: der *Rohertrag* läßt sich zwar aus den beiden anderen Merkmalen bestimmen, aber wenn man den *Verkaufspreis* aus einem (geplanten) *Rohertrag* und dem (schon bezahlten) *Einkaufspreis* bestimmen will, benötigt man eine weitere Funktion. Im Extremfall muß man hier also drei im wesentlichen redundante Funktionen spezifizieren, ohne daß man an Verständnis oder Struktur für die Anwendung gewonnen hätte.

Daher plädieren wir dafür, alle drei Eigenschaften als Merkmale zu spezifizieren und den Zusammenhang, in dem sie stehen, als Zusicherung (*constraint*). Aus fachlicher Sicht ist das völlig ausreichend und oftmals auch viel zutreffender als die Lösung mit Funktionen (Symmetrie!); jede Eigenschaft ist als Merkmal ansprechbar, und man kann sich darauf verlassen, daß die Merkmalsbelegung konsistent ist (denn die

---

[2] von: Animismus: Glaube an die Allbelebung der Natur

[3] Das Klammerpaar deutet wie in C an, daß es sich um eine Fähigkeit handelt.

Spezifikation verpflichtet den Programmierer, dafür zu sorgen, daß die Zusicherung nie verletzt wird).

Merkmale und Typen

Der Inhalt eines Merkmals eines Objekts kann entweder ein Wert sein oder ein Verweis auf ein anderes Objekt.[4] Ist der Inhalt ein Wert, so kann man, wie in modernen Programmiersprachen üblich, von vornherein sagen, von welchem *Datentyp* dieser Wert ist. Der Datentyp ist die Menge der möglichen Werte, die das Merkmal annehmen kann. So ist der Datentyp des Merkmals *Alter* einer *Person* zum Beispiel *_Integer*[5] oder noch besser *_Lebensalter*. *_Lebensalter* ist dann möglicherweise ein[6] Untertyp *[0 .. 130]* von *_Integer*.

Analog ordnen wir allen "gleichartigen Objekten" einen *Objekttyp* zu. Gleichartig heißen zwei Objekte, wenn sie dieselben Eigenschaften haben, also dieselben Merkmale und dieselben Fähigkeiten. Dabei müssen gleichartige Objekte natürlich nicht dieselben Merkmals*werte* haben. Allerdings müssen die sich entsprechenden Merkmale denselben Typ haben.

Ohne es bislang erwähnt zu haben, haben wir Objekttypen in obigen Beispielen schon verwandt. Wir sprachen immer von den Eigenschaften einer *Person.* Es war uns aber egal, über welche Einzel-*Person* wir sprachen, denn die beschriebenen Eigenschaften galten für alle *Personen*. So hat **jede** *Person* die Merkmale *Vater* vom Typ "Verweis auf Person" oder *Alter* vom (Daten-)Typ *_Lebensalter,* und prinzipiell kann jede *Person wählen()*. Die einzelnen Personen unterscheiden sich erst durch aktuelle Belegung der Merkmale, also durch den Wert des *Alters* oder den Verweis auf den *Vater,* also eine bestimmte *Person.* Daher hätten wir korrekterweise bisher "*Person"* mit Unterstrich schreiben müssen ("*_Person"*), denn das ist der Name des Objekttyps.

Datentypen

Wir kennen in der Spezifikation einerseits die gebräuchlichen elementaren Datentypen wie *_Integer*, *_Real*, *_String*, usw.; andererseits stehen die üblichen Sprachmittel zur Verfügung, um sich neue Datentypen aus bereits definierten zu erzeugen. Diese Sprachmittel sind insbesondere die Strukturbildung (*record*), die Aufzählung (*enumeration*), die Einschränkung (*subtype*) und die erweiterte Form eines (dynamischen) Feldes (*array*). Zusätzlich kann man einem Datentyp Routinen zuordnen, die Plausibilitäten prüfen. Zum Beispiel kann man die *ISBN* als eine Datenstruktur spezifizieren, bestehend aus *Landnummer*, *Verlagsnummer*, interner *Buchnummer* und *Prüfziffer*. Eine Plausibilitätsprüfung wird dann sicherstellen, daß die

---

[4]Diese Aussage werden wir weiter unten noch präzisieren (siehe den Abschnitt *Umfassung*).

[5]Um Datentypname von Merkmalsnamen zu unterscheiden, schreiben wir vor jeden Typnamen einen Unterstrich.

[6]Wir schreiben hier "*ein Untertyp*", da es weitere Untertypen *[0 .. 130]* von *_Integer* gibt wie zum Beispiel *_Omnibusgeschwindigkeit.*

gewichtete Quersumme durch 11 teilbar ist, da es sich sonst nicht um eine gültige ISBN handelt.

Objekttypen

Objekte, d.h. Exemplare desselben Typs haben dieselben Merkmale und Fähigkeiten. Verschiedene Objekte eines Typs unterscheiden sich nur in der Belegung (den Werten) der Merkmale[7]. Die Spezifikation der Merkmale und Fähigkeiten sowie des Lebenslaufs legen wir demzufolge in der *Objekttypbeschreibung* nieder. Aus programmtechnischer Sicht kann man einen Objekttyp als abstrakten Datentyp auffassen, also als Datenstruktur (record) mit darauf operierenden Funktionen. In Ada würde man typischerweise diese Datenstruktur und die Funktionen zu einem *package* zusammenfassen.

Datentyp oder Objekttyp?

Der Übergang von Daten- zum Objekttyp ist fließend und abhängig von der Anwendung, die spezifiziert wird. Das *_Datum* ist normalerweise eine Datenstruktur, bestehend aus *Tag*, *Monat* und *Jahr*. Ist die zu spezifizierende Anwendung allerdings ein Zeitmanagementsystem, so ist vorstellbar, das man ein *Datum* als Objekt auffaßt.

Folgende Kriterien geben Hinweise auf einen Objekttyp:

- Kann man ein identifizierendes Merkmal benennen? Es sollte in der Realität vorhanden und beim Anwender in Gebrauch sein. Ein technischer Schlüssel ist hier nicht gemeint. Das identifizierende Merkmal kann auch zusammengesetzt sein; z.B. kann man ein *Buch* eindeutig identifizieren mit den Merkmalen *Landnummer*, *Verlagsnummer* und *Buchnummer*, denn das sind genau die relevanten Bestandteile der ISBN.

- Drängt es den Anwender, zu einem neu angelegten Objekt (im zu realisierenden System) weitere Informationen zu sammeln, also die restlichen Merkmale mit Werten zu versehen?

- Machen Operationen für Anlegen und Löschen eines Objekts Sinn? Oder möchte man stattdessen lieber die Informationen *eintragen?* Dann handelt es sich eher *nicht* um ein Objekt. Beispielsweise möchte man das *Alter* einer *_Person* typischerweise nicht "anlegen", sondern "eintragen"; also wird *Alter* sinnvollerweise nicht als Objekt modelliert, sondern als Merkmal einer *_Person.*

- Hat das Objekt Fähigkeiten, die über Plausibilitätsprüfungen hinausgehen?

---

[7]Es gibt Methoden, in denen zwei Objekte eines Typs unterscheidbar sind, obwohl sie in allen Merkmalswerten übereinstimmen (siehe z.B. [Meyer88]. Wir denken, man sollte in diesem Fall ein weiteres Merkmal einführen, das Objekte eines Typs eindeutig identifiziert (Schlüssel).

- Hat das Objekt einen Lebenslauf?

Diese Kriterien sind aber keine Definition für ein Objekt, sie müssen also nicht alle zwingend mit "Ja" beantwortet werden, um zu entscheiden, ob es sich um ein Objekt handelt.

Identifizieren von Objekten

Fast immer kann der Anwender zu jedem spezifizierten Objekt ein Merkmal nennen oder manchmal auch eine Merkmalskombination, mit dem sich ein Objekt eindeutig identifizieren läßt. Ein *_Wertpapier* wird eindeutig identifiziert durch die *WertpapierKennummer*, eine *_Person* durch *Vorname, Name* und *Anschrift*, ein *_PKW* durch sein *PolizeilichesKennzeichen* usw. Wir gehen noch etwas weiter, und verlangen zwingend, daß man zu jedem Objekttyp ein Merkmal oder eine Merkmalskombination angibt, deren Wert ein Exemplar eindeutig unterscheidet von jedem anderen Exemplar dieses Objekttyps.

Möglich ist auch, daß man mehrere Alternativen zur eindeutigen Identifizierung eines Objekts spezifizieren will oder muß: ein *_PKW* wird zusätzlich zum oben angegebenen Merkmal auch durch das Merkmal *FahrgestellNummer* eindeutig identifiziert, und es sind Anwendungen denkbar, in denen beide Identifikatoren notwendig sind: so muß die Polizei ein gestohlenes Auto anhand der *FahrgestellNummer* identifizieren, und Rotlichtsünder bekommen ihren Bußgeldbescheid, indem das *PolizeilicheKennzeichen* ausgewertet wird.

Zustände und Zustandsraum

Datentypen beschreiben Wertemengen. Das karthesische Produkt der Wertemengen aller Merkmale eines Objekttyps ist der *Zustandsraum* dieses Objekttyps. Jedem Objekt wird ein Punkt im Zustandsraum zugeordnet, der sich aus der Belegung der Merkmale des Objekts ergibt. Als *Zustandsvektor* bezeichnen wir demzufolge den Vektor zu einem Objekt, dessen Koordinaten gerade die Merkmalswerte des Objekts sind[8]. Jedem Objekt ist damit eindeutig ein Zustandsvektor zugeordnet. Durch jede Änderung eines Merkmalwertes ändert sich der Zustandsvektor dieses Objekts, d.h. dem Objekt wird ein neuer Punkt im Zustandsraum zugeordnet.

Umgekehrt gibt es zu jedem Punkt im Zustandsraum (also zu jedem möglichen Zustandsvektor) höchstens ein korrespondierendes Objekt zu einem festen Zeitpunkt. Das liegt daran, daß wir zwingend fordern, daß zu jedem Objekttyp ein Merkmal oder eine Merkmalskombination spezifiziert ist, die ein Objekt eindeutig identifizieren. Je zwei Exemplare eines Objekttyps müssen sich im Wert dieses Merkmals oder dieser Merkmalskombination unterscheiden.

---

[8] Aus diesem Grund bezeichnet man Merkmale oft als *Zustandskomponenten* des Objekts.

Die Begriffe Zustandsraum und Zustandsvektor sind wichtig zur theoretischen Fundierung der hier vorgestellten Konzepte. In der Welt des Anwenders ist dagegen der im folgenden beschriebene Begriff des Zustands wichtig:

Jedem Objekt ordnen wir implizit ein Merkmal *Zustand* zu. Der *Zustand* bestimmt sich aus den anderen Merkmalswerten, aber er ändert sich nicht zwangsläufig, wenn sich der Wert eines anderen Merkmals ändert. Ein Objekt *Person* kann zum Beispiel im Zustand *volljährig* sein, sobald der Wert des Merkmals *Alter* 18 überschritten hat. Eine Änderung des Merkmals *Körpergröße* ändert zwar den Zustandsvektor der Person, nicht jedoch den Zustand *volljährig*.

In den Zustandsraum eines Objekttyps abgebildet, entspricht der Zustand einer *Menge* von Zustandsvektoren. Gehört der Zustandsvektor eines Objekts zu solch einer Menge, dann ist das Objekt im korrespondierenden Zustand. Die Mengen, die den Zuständen eines Objekts entsprechen, müssen disjunkt sein.

Um Verwechslungen zu vermeiden, sei hier darauf hingewiesen, daß [Meyer88] nur den Zustandsvektor in unserem Sinn kennt; [Simonsmeier90] verwendet dafür in seiner deutschen Übersetzung das Wort "Zustand".

Lebenslauf

Die Zustände sowie die zustandsverändernden Fähigkeiten des Objekts kann man in einem endlichen Automaten, dem *Lebenslauf,* darstellen (Abb. 2). Die Knoten entsprechen den Ausprägungen des Zustands, die Kanten den zustandsverändernden Fähigkeiten. Darüber hinaus enthält der Lebenslauf auch die Fähigkeiten, die nur in bestimmten Zuständen des Objekts nutzbar sind, aber nicht notwendig den Zustand verändern, oder die, abhängig von einer Rückmeldung, zu verschiedenen Zuständen führen können. Zum Beispiel kann man ein *Buch* erst dann *verkaufen(),* wenn es *lieferbar* ist, und wenn man das letzte verkauft hat, dann ist es *ausverkauft*. Fähigkeiten wie *verkaufen()* zeigen sich im Graphen des Lebenslaufs als Kanten, die in den selben Zustand zurückkehren bzw. als "gespaltene Pfeile", die von einem Zustand in mehrere andere führen.

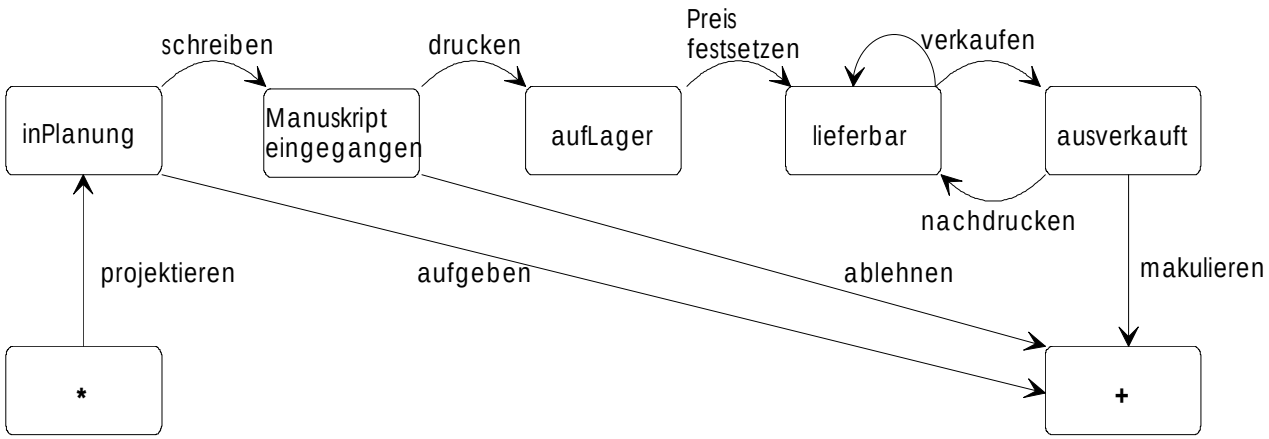

Abb. 2 Graph des Lebenslauf eines *Buches*

Beziehungen zwischen Objekten

Objekte stehen miteinander in Beziehung. Es gibt vier Arten von Beziehungen: ein Objekt *benutzt* ein anderes Objekt (genauer: es nutzt eine Fähigkeit des anderen Objekts), ein Objekt *liest* Merkmalswerte eines anderen Objekts, ein Objekt *spezialisiert* ein anderes Objekt oder ein Objekt *gehört* (zu) einem anderen bzw. das andere Objekt *umfaßt* das eine.

Nutz- und Lese-Beziehung

Die Lese-Beziehung ist ein Spezialfall der Nutz-Beziehung (*interacts-with-Relation*), denn das Lesen eines Merkmalswertes eines anderen Objekts ist in Wirklichkeit die Benutzung der (impliziten, gleichnamigen) Fähigkeit des anderen Objekts, einen Merkmalswert auszulesen. Um ganz exakt zu sein, ist ein Merkmal der Spezialfall einer Fähigkeit, nämlich die Projektion des Zustandsvektors auf eine Zustandskomponente.

Damit ein Objekt ein anderes nutzen (d.h. insbesondere dessen Merkmalswerte lesen) kann, muß man eine Beziehung in Form eines Verweises zum genutzten Objekt herstellen. Dies geschieht ebenfalls über Merkmale. So könnte eine *_Person* außer den Merkmalen *Alter* und *Zustand* auch die Merkmale *Vater* und *Mutter* haben. Die Werte dieser Merkmale sind Verweise auf (andere) *_Personen*. Um die Eigenschaften eines Objekts, auf das man einen Verweis hat, zu nutzen, verwenden wir die Punktschreibweise: das Alter der Mutter bekommt man mit *Mutter.Alter*, und mit *Vater.wählen()* kann man den Vater zum Wählen schicken. Diese Punkt-Notation kann man rekursiv anwenden, so daß beliebig lange Merkmal-Punkt-Merkmal-Folgen entstehen[9]: das Alter des Großvaters mütterlichseits ist *Mutter.Vater.Alter*. Das Verfolgen der Merkmalsverweise nennt man *Dereferenzieren*. Dereferenziert man das Merkmal *Mutter*, so "befindet man sich bei" einem Exemplar der *_Person*, nämlich der *Mutter*. Dereferenziert man *Mutter.Vater*, so befindet man sich beim Großvater. Dereferenziert man *Mutter.Vater.Alter*, so erhält man das Alter des Großvaters.

Zusätzlich zu den Merkmalen, die Daten enthalten, können auch auch verweisende Merkmale Bestandteil einer identifizierenden Merkmalskombination sein. Die Vorteile dieses Konzepts wurden ausführlich in [Ritterbach93] diskutiert.

Umfassung (Inbesitznahme)

Von ganz anderer Natur als die eben beschriebene Beziehung zu einem Objekt ist die Beziehung zu einem Datentyp. Während auf ein Exemplar eines Objekttyps (also ein

---

[9]Selbstverständlich müssen alle "inneren Merkmale" der Punktnotation, also alle Merkmale, die *vor* einem Punkt stehen, Verweise auf ein anderes Objekt sein.

Objekt) beliebig viele Verweise zeigen können, und somit der Wert eines Merkmals dieses Objekts praktisch öffentlich zugänglich ist, existiert das Exemplar eines Datentyps nur im Zusammenhang mit dem Objekttyp. So kann sinnvollerweise jedes Objekt eine Beziehung aufnehmen zum Exemplar (zum Objekt) "Meyer" des Typs *_Person*. Doch was soll eine Beziehung sein zum Exemplar "18" des Datentyps *_Integer?* Und wenn mehrere Objekte eine Beziehung zur "18" aufnehmen, was bedeutet das dann?

Man kann sich diese Art der Beziehung folgendermaßen vorstellen: Ein Objekt (z.B. *_Person*) kann eine "Beziehung" zu einem Exemplar eines Datentyps (z.B. *_Integer* "18") aufnehmen über ein Merkmal (z.B. *Alter),* indem das entsprechende Exemplar kopiert wird und dann vom Objekt in Besitz genommen. Diese Kopie des Datentypexemplars gehört daraufhin dem Objekt (das Objekt *umfaßt* das Exemplar) und nur über das umfassende Objekt kann man Bezug nehmen auf das kopierte Datentypexemplar (hier "18"). Innerhalb des Systems kann es also eine ganze Reihe von identischen Datentypexemplaren geben, die dadurch unterscheidbar sind, daß sie von verschiedenen Objekten umfaßt werden.

Diese Art der Beziehung kann man ausdehnen auf Objekttypen: ein Objekttyp kann einen anderen in Besitz nehmen und umfassen, d.h. nur über den umfassenden Objekttyp kann Bezug genommen werden auf Eigenschaften des umfaßten Objekttyps. So umfaßt beispielsweise ein *_Auftrag* seine *_Auftragspositionen.* Diese Beziehung nennt man auch oft *Aggregation* oder *Objekterweiterung*.

Schließlich wollen wir auch die letzte Typkombination eines Merkmals zulassen, nämlich den *Datentypverweis*.

In Spezifikationen kommen die beschriebenen Beziehungen unterschiedlich häufig vor:

Beziehungsarten	**Datentyp**	**Objekttyp**
Umfassung	sehr häufig	manchmal
Verweis	selten	häufig

Spezialisierung (Vererbung)

in der Spezifikationsphase (und nicht nur da) erlebt man immer wieder, daß man einen Sachverhalt mehrfach aufschreiben bzw. die entsprechenden Textstellen kopieren müßte. Das passiert meist bei Objekten, die sich ähnlich sind oder sogar eine "gemeinsame Abstammung" haben, also Spezialisierungen eines gemeinsamen Vorfahren sind. Dieser Vorfahr ist fast immer ein Objekt, das in der Welt des Anwenders

bereits existiert. So handeln Banken mit Aktien und Renten oder allgemeiner mit Wertpapieren.

Um solche Redundanzen zu vermeiden, nutzen wir das Konzept der *Vererbung (is-a-Relation)*. Wir notieren also alle gemeinsamen Eigenschaften von Renten und Aktien bei dem Objekttyp *_Wertpapier*. Beispiele für solche Eigenschaften sind das Merkmal *WertpapierKennummer* oder die Fähigkeit *ErtragBerechnen()*. Da sowohl *_Aktien* als auch *_Renten* die Eigenschaften des *_Wertpapiers* erben, besitzen sie die Merkmale und Fähigkeiten, die beim *_Wertpapier* spezifiziert wurden, wie "eigene", d.h. man kann normalerweise von außen nicht erkennen, ob eine Eigenschaft bei diesem Objekttyp oder bei einem Vorfahr spezifiziert wurde.

Somit ist die hier definierte Spezialisierungsbeziehung eine Beziehung auf Beschreibungsebene bzw. Objekttypebene, auch wenn wir weiter oben von einer Objektbeziehung sprachen.

Vorgeschriebene Eigenschaften und "dynamisches Binden"

Eine Besonderheit der Vererbung ist, daß sie Objekttypen erlaubt, Eigenschaften *vorzuschreiben* für ihre Nachfahren (*deferred* bzw. *virtual features*). Spezifiziert man zum Beispiel beim *_Wertpapier* die Fähigkeit *ErtragBerechnen()* als vorgeschrieben, so müssen sowohl die *_Renten* als auch die *_Aktien* eine Realisierung[10] dieser Fähigkeit anbieten[11]. Typischerweise ist das Vorschreiben einer Eigenschaft verbunden mit einer Umbenennung. So würde man die vorgeschriebene Fähigkeit *ErtragBerechnen()* bei der *_Aktie* als *DividendeBerechnen()* bezeichnen und bei der *_Rente* als *ZinsBerechnen()*.

Ein weiteres Konzept der Vererbung im Zusammenhang mit vorgeschriebenen Fähigkeiten ist das d*ynamische Binden*, also die Bestimmung des Daten- oder Objekttyps erst zur Laufzeit des Systems. Dynamisches Binden ist also eigentlich ein programmtechnisches Konstrukt, doch es läßt sich auch in der Spezifikation gut nutzen, wie dieses Beispiel zeigt: Ein Objekt *_Depot* könnte die Fähigkeit *EinlagenBewerten()* haben, deren Spezifikation folgende Zeilen enthalten könnte:

> Für jedes *Wertpapier* im *Depot*
> *Wertpapier.ErtragBerechnen*(*Ertrag*)
> *Summe = Summe + Ertrag*

Die vorgeschriebene Fähigkeit *ErtragBerechnen()* muß also (zur Laufzeit) erkennen, ob das Wertpapier eine *_Rente* oder eine *_Aktie* ist und dementsprechend den *ZinsBerechnen()* oder die *DividendeBerechnen()*.

---

[10] "Realisierung" nicht im Sinne eines Programms, sondern im Sinne einer Beschreibung des Effekts der Fähigkeit, s.u.

[11] Die Realisierung der Fähigkeit wurde also *aufgeschoben*: Sie wurde nicht schon beim *_Wertpapier*, sondern erst bei der *_Rente* bzw. *_Aktie* spezifiziert. [Simonsmeier90] nennt dies deshalb *aufgeschobene Routine*.

Vererbung von Lebensläufen

ist ein schwieriges Thema. Einfach ist es, wenn ein Objekttyp keinen eigenen Lebenslauf hat: dann gilt nämlich der Lebenslauf des nächsten Vorfahren, der einen (expliziten) Lebenslauf hat. Kompliziert ist es, wenn ein Objekt einen eigenen Lebenslauf hat, aber auch ein Vorfahr. Es gibt einen sehr theoretischen Ansatz, der besagt, daß der Lebenslauf eines Nachfahren eine Verfeinerung des aktuellen Lebenslaufs sein muß. Dieser Ansatz engt häufig den sehr erwünschten kreativen Gedankenfluß des Spezifikateurs ein, da dieser versuchen muß, den Lebenslauf eines Vorfahren so zu verfeinern, daß er möglichst gut zu seinem Objekttyp paßt, anstatt den fachlichen Lebenslauf, frei von allen formalen Gesichtspunkten, so zu notieren, wie er der Wirklichkeit am ehesten entspricht.

Wir tendieren dazu, für jeden Objekttyp den fachlich korrekten Lebenslauf zu notieren. Damit gelten *gleichzeitig* dieser Lebenslauf und die Lebensläufe aller Vorfahren dieses Objekttyps. Allerdings muß nun ein Zustandsübergang in einem Vorfahr-Lebenslauf von einer Fähigkeit explizit ausgelöst werden, wobei dies typischerweise dadurch geschieht, daß man eine zustandsverändernde Fähigkeit des Vorfahren nutzt. Deshalb durchläuft auch jedes Objekt den grundlegenden Lebenslauf des *_Ding*, also des Vorfahren eines jeden Objekttyps. Dieser Lebenslauf hat die drei Zustände "*", "existiert" und "+". Alle Fähigkeiten, die ein Objekt *erzeugen*, führen es aus dem Zustand "*" in den Zustand "existiert", alle Fähigkeiten, die ein Objekt *vernichten*, führen von "existiert" zu "+".

Fähigkeiten, die Objekte eines Typs nur gemeinsam anbieten können

Es gibt Merkmale, die man nicht einem einzelnen Objekt zuordnen kann, und es gibt Fähigkeiten, die nur die Gesamtheit der Objekte eines Objekttyps leisten können. Diese Eigenschaften heißen *Typmerkmale* beziehungsweise *Typfähigkeiten.* Ein Typmerkmal ist zum Beispiel die aktuelle oder auch die maximale Anzahl von Objekten eines Typs, eine Typfähigkeit ist die Fähigkeit, die Objekte nach einem Merkmalswert zu sortieren.

Die Typmerkmale kann man nicht in den Zustandsraum abbilden, da es sich um Daten *über* den Zustandsraum und nicht *innerhalb* des Zustandsraums handelt. Somit muß es eine übergeordnete Einrichtung geben, die diese Merkmale verwaltet und die Typfähigkeiten anbietet. Wir stellen uns zu jedem Objekttyp einen *Verwalter* vor, der diese Aufgabe übernimmt.

Um eine *Typeigenschaft*, also eine Eigenschaft des Typverwalters anzusprechen, verwenden wir wiederum die Punktnotation, wobei diese jetzt an einer Stelle statt eines Merkmalsnamens einen Objekt*typ*namen enthält. So wird zum Beispiel die aktuelle Anzahl der *_Konto*'s mit *_Konto.Anzahl* bezeichnet, und es könnte die Typfähigkeit *_Konto.sortieren()* geben.

3 Eine Notation für Objekttypen

In diesem Abschnitt stellen wir eine Notation zur Spezifikation von Objekttypen vor. Es wird gezeigt, wie die eben beschriebenen Konstrukte in einer Sprache formuliert werden können. Diese Sprache entstand zum Großteil in laufenden Projekten und wurde vielen Änderungen und Ergänzungen unterworfen, doch der jetzige Wortschatz ist recht stabil. Die zugrundegelegte Semantik erfüllt nicht die Kriterien, die man an eine Programmiersprache stellt, und soll es auch nicht können. Trotzdem wollen wir diese Semantik möglichst exakt festzulegen und zugleich formalen Ballast vermeiden. Die folgende Notation soll dem Spezifikateur helfen, einen komplexen Sachverhalt strukturiert und übersichtlich zu formulieren und ihn nicht in das enge Korsett einer zu formalen Syntax zwängen.

Abb. 3 zeigt ein typisches Beispiel[12] einer Objekttypbeschreibung. Um in der Diskussion auf einzelne Abschnitte bezugnehmen zu können, wurden viele Zeilen vorne mit einer Nummer versehen; sie sind kein Bestandteil der Objekttypbeschreibung.

```
01 JEDE          _Buchung
02 IST_EIN       _Vertrag

03               Die ^_Buchung verbindet eine Reise (also das Katalogangebot)
                 mit einem Teilnehmer und mindestens drei Reservierungen
                 einer Leistung. Diese drei Leistungen nennt man das
                 Touristische Tripel. [...]

04 HAT           [Buchungs]Nummer          :     _BuchungsNr
05 HAT           zugrundegelegtesAngebot   --->  _ReiseAngebot
06 HAT           Reiseort                  --->  _Ort
   HAT           Reisezeit                 --->  _Zeitraum
07 HAT           Teilnehmer                ::    _Teilnehmer
08 ??            Gehören Teilnehmer nicht besser zur Reservierung      ??
09 !!            Nein, denn man muß ihnen z.B. die Reiseunterlagen schicken
                 oder wissen, wer gebucht hat      !!
10 HAT           Reservierung              ::    _Reservierung
11 PRUEFEN       Eine ^_Reise umfaßt mindestens drei ^_Reservierungen.
12 HAT           Preis                     o:    _DM-Betrag
   PRUEFEN       ^Preis = Summe aller ^Reservierung.Preis
                 Der ^Preis ist nicht mehr optional ab Zustand ^abgeschlossen.
```

---

[12] Dieses Beispiel ist hier nicht vollständig wiedergegeben und daher nicht konsistent. Es ist der Ausschnitt aus einer parallel zu diesem Papier entstehenden Arbeit, die einen Teil des touristischen Buchungssystems aus [Denert91] konsistent spezifiziert.

13 FINDEN_MIT *BuchungsNummer*
Teilnehmer + Reisezeit

14 LEBENSLAUF

*	anlegen	inBearbeitung
inBearbeitung	freigeben	freigegeben
inBearbeitung	freigeben	inBearbeitung
freigegeben	UnterlagenProduzieren	abgeschlossen
abgeschlossen	<<ändernde Fähigkeit >>	wiederaufgenommen
wiederaufgenommen	freigeben	wiederfreigegeben
wiederfreigegeben	UnterlagenProduzieren	abgeschlossen
inBearbeitung	stornieren	+
freigegeben	stornieren	+
abgeschlossen	stornieren	+
wiederaufgenommen	stornieren	+
wiederfreigegeben	stornieren	+

15 -- *Beachte, daß ^UnterlagenProduzieren() abhängig vom ^Zustand bereits produzierte Unterlagen berücksichtigen muß!* --

16 ENTSTEHT_DURCH ***anlegen***
17 AENDERT *AbwickelndeAgentur*
StartOrt
Reisezeit
Teilnehmer
18 EFFEKT *_Buchung.NummerVergeben(BuchungsNummer)*
^*Buchungstag* = Systemdatum
19 ENDE(ok,inBearbeitung)

20 ERLISCHT_DURCH ***stornieren***
EFFEKT alle Reservierungen rückgängig machen (Anbieter benachrichtigen)
alle bereits produzierten Unterlagen einziehen und dem Altpapier zuführen
ENDE(ok,+)

21 KANN_MAN ***freigeben***
22 BRAUCHT *Sachbearbeiter* : _**Mitarbeiter**
23 ANNAHME *^Sachbearbeiter.ArbeitsStelle = ^AbwickelndeAgentur*
-- *Die Buchung darf nur von Sachbearbeitern der abwickelnden Agentur freigegeben werden.* --
EFFEKT Die für die **^_Buchung** reservierten Leistungen werden auf Konsistenz geprüft, und zwar bezüglich folgender Fragen:
- Ist für alle ^*Reservierung*en ^Reservierung.bestätigt?
sonst ENDE(ReservierungNichtBestätigt,inBearbeitung)

- Passen Flüge und Hotels zeitlich und räumlich zusammen, d.h. liegt zu jedem Zeitpunkt das Hotel, zu dem die gerade reservierten Zimmer gehören, in dem Gebiet, in das zuletzt geflogen wurde?
  sonst ENDE(FlugHotelInVerschGebieten,inBearbeitung)
  [...]
  da jetzt die Buchung konsistent ist, ENDE(ok,freigegeben)

```
    --          ------------------------------------------------------------------------ --
24  DER         _Buchung.Verwalter
25  HAT         letzteVergebeneNummer          :     _BuchungsNr
26  KANN        [Buchungs]NummerVergeben
27  LIEFERT     Nummer                         :     _BuchungsNr
    EFFEKT      ...


    --          ------------------------------------------------------------------------ --
28  DER         _Zeitraum
    BESTEHT_AUS von                            :     _Datum
                bis                            :     _Datum
29  PRUEFEN     von <= bis
```

Abb. 3 Beispiel für eine OT-Spezifikation

Auf den ersten Blick mag diese Notation zumindest gewöhnungsbedürftig sein. Doch sie entspricht (und entspringt) unserem obersten Prinzip, nämlich der Verwendung einer möglichst klaren und guten Sprache in der Spezifikation, dem Ringen um eindeutige und präzise Formulierungen. Nicht der Leser soll sich quälen, den Text zu verstehen, sondern der Autor muß sich ganz besonders anstrengen, den Text leicht verständlich zu gestalten.

Das Wort "JEDE"

Die Objekttypbeschreibung beginnt mit dem Wort "JEDE[R/S]" [01][13] und der Benennung des Objekttyps sowie der Einordnung in die Vererbungshierarchie [02]. Um zum Ausdruck zu bringen, daß die hier beschriebenen Eigenschaften (Merkmale, Fähigkeiten, Lebenslauf) für **alle** Objekte eines Typs gelten, beginnt die Beschreibung mit einem der Schlüsselwort "JEDE/JEDER/JEDES": *Jedes* Objekt *hat* das Merkmal oder *kann* diese Fähigkeit anbieten.

Auch im Zusammenhang mit der Vererbungsbeziehung ist uns das Wort "jedes" sehr wichtig. Es mag zum Beispiel folgendes gelten: die *_Person* "Meyer" *ist ein* Mitglied

[13] Mit einer Nummer in eckigen Klammern nehmen wir Bezug zur entsprechenden Zeile in Abb. 3.

des Vereins zur Pflege der objektorientierten Sprache. Deswegen ist aber sicherlich nicht *jede _Person* dort Mitglied. Bei der Spezialisierung gilt hingegen, daß *jedes* Objekt eines Objekttyps auch Objekt eines jeden Vorfahr-Objekttyps ist.

Lehrbuchhafte Beschreibung des Objekttyps

Es folgt eine ausführliche *fachliche* Beschreibung des Objekts `[03]`. Sie soll etwa wie ein entsprechendes Kapitel eines Lehrbuchs des Anwendungsgebiets aufgebaut sein, so daß die Sammlung der Beschreibungsabschnitte der Objekttypen dem Leser helfen, zum Experten der Anwendung zu werden.

Spezifikation der Merkmale

Anschließend werden die Merkmale spezifiziert, die jedes Objekt dieses Objekttyps *hat* `[04-12]`. Eine Merkmalsspezifikation besteht aus bis zu fünf Teilen:

Die erste Zeile `[04]` enthält den Merkmalsnamen, die Art der Beziehung und den Objekt- bzw. Datentyp des Merkmals. An Beziehungsarten unterscheiden wir die Inbesitznahme und den Verweis und drücken sie durch einen Doppelpunkt bzw. durch einen Pfeil aus, unabhängig davon, ob auf einen Datentyp oder einen Objekttyp Bezug genommen wird. So sind in dieser Spezifikation *_Ort* `[06]` und *_BuchungsNr* `[04]` Datentypen und *_ReiseAngebot* `[05]` und *_Reservierung* `[10]` Objekttypen, auf die in jeweils verschiedener Art Bezug genommen wird. Auf *_Ort* und *_ReiseAngebot* wird verwiesen, d.h. auch andere Objekte (auch eines anderen Objekttyps) können zu ihnen unmittelbare Beziehungen aufbauen. Die *_BuchungsNr* und die *_Reservierungen* hingegen werden umfaßt und die Exemplare, die eine *_Buchung* in Besitz genommen hat, sind nur über diese *_Buchung* erreichbar (Abb. 4).

Mit der Beziehungsart kann man, wie man in `[07]`, `[10]` und `[12]` sieht, auch Kardinalitäten ausdrücken: der zweifache Doppelpunkt in `[12]` heißt, daß eine *_Buchung* nicht nur eine, sondern mehrere (genau: mindestens eine) *Reservierungen* umfaßt. Ein "o" in der Angabe der Beziehungsart in `[12]` steht für "optional", so daß die Umfassung von keinem, einem oder mehreren Objekt- bzw. Datentypen mit "o::" spezifiziert werden kann. Die Kardinalitätsangaben bei Verweisen schlagen sich in der Anzahl der Pfeilspitzen bzw. einem "o" im Schaft nieder, so daß zum Beispiel die Notation für einen Verweis auf kein, ein oder mehrere Objekt- bzw. Datentypen "--o>>" wäre.

Abb. 4 zeigt an einen winzigen Ausschnitt des Objekttypbildes, wie man diese Beziehungsarten auch graphisch darstellen kann.

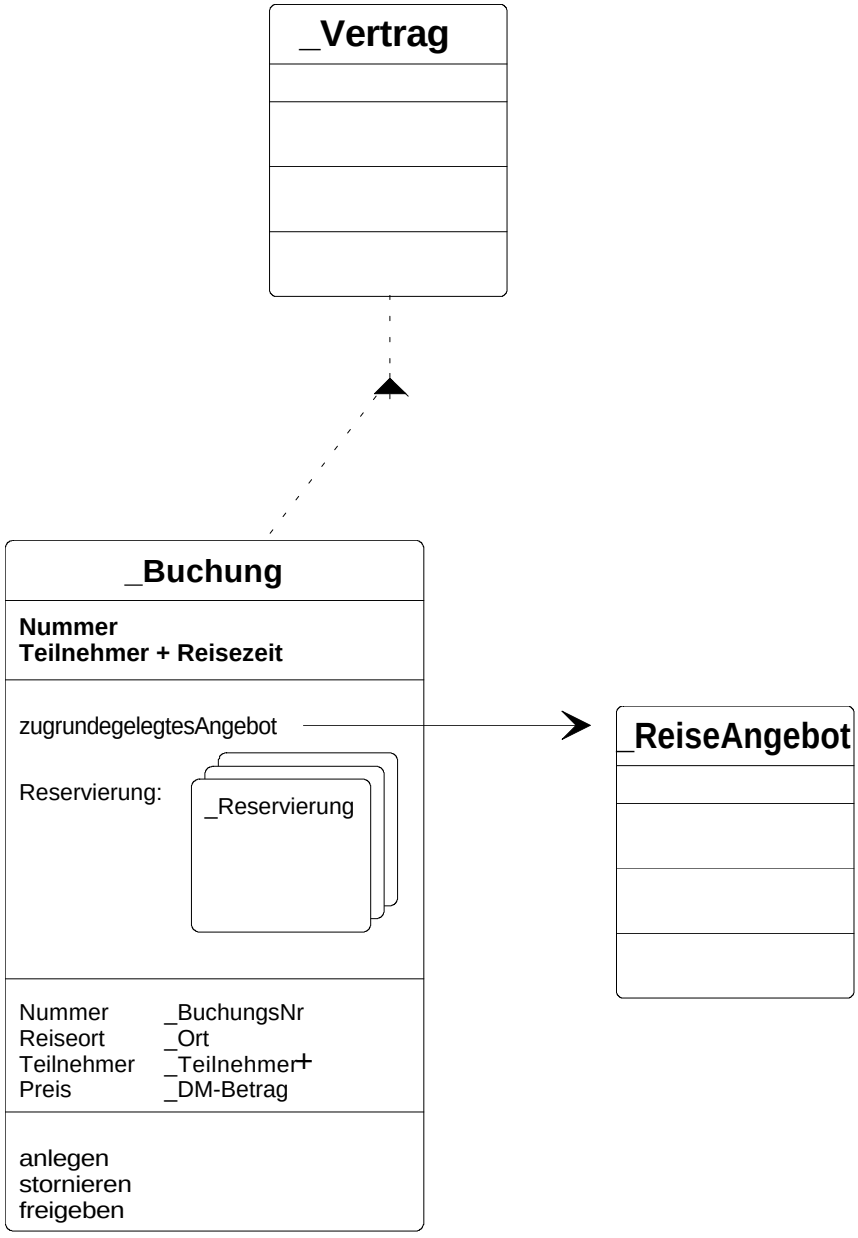

Abb. 4 Ein Ausschnitt des Objekttypbildes

Der vierte Teil einer Merkmalsspezifikation ist die optionale Beschreibung. Diese ist deswegen optional, weil sie oft überflüssig ist, wenn nur der Merkmalsname sorgfältig gewählt wurde. So wäre eine typische Beschreibung für das Merkmal *Reisezeit:* "Zeitraum, in dem die Reise stattfindet bzw. stattfinden soll". Diese Beschreibung enthält keine zusätzlichen Informationen, sondern bläht die Spezifikation auf und ermüdet den Autor, sollte dieser gezwungen sein, zum Merkmal eine Beschreibung zu finden, genauso wie den Leser. Erfahrungsgemäß leidet durch diese Ablenkung auch die Qualität des Merkmalsnamen.

Schließlich kann man zu einem Merkmal Zusicherungen oder Plausibilitätsprüfungen spezifizieren [11], die nicht den Merkmals**typ** betreffen (siehe [29]), sondern das

Zusammenwirken des Merkmals mit anderen (vergleiche den Abschnitt *Merkmale oder Fähigkeiten*).

Querschnittskonzepte für Dokumente

An dieser Stelle wollen wir auf drei Konstrukte hinweisen, die in obigem Beispiel zur Spezifikation von Merkmalen benutzt wurden, aber nicht nur hier, sondern an *jeder* Stelle der Objekttypbeschreibung verwendet werden können.

Zum einen ist es das *Caret* "^", mit dem die Namen von Merkmalen, Fähigkeiten, Daten- und Objekttypen und Zuständen markiert werden dürfen und sollen (`[03]` oder `[11]`). Unsere Werkzeuge erkennen sie dann auch in frei formatierten Texten (wie Beschreibungen) und können zum Beispiel überprüfen, ob die Namen gültig sind, also ob zum Beispiel die Dereferenzierung der Punktnotation möglich ist.

Zum anderen sind es *Kommentare*, die, abhängig von der Kommentarart, in doppelte "-", "?" oder "!" eingeschlossen sind (`[08]`, `[09]` oder `[15]`). Ein mit "--" markierter Kommentar ist ein herkömmlicher Kommentar, wie er von Programmiersprachen her bekannt ist. Vor allem in formalen Abschnitten wie dem Lebenslauf notiert man hier weitere, meist kurze Erläuterungen und Verdeutlichungen.

Beim Spezifizieren entstehen oft Fragen, die man nicht sofort klären kann, oder offene Enden, zu denen man noch einen Gedanken notieren will. Dazu dienen die Kommentararten "Fragen" und "Antworten". Gerade in den Fragen und Antworten schlägt sich der Entwicklungsprozeß des Dokuments deutlich nieder, und der spätere Leser wird möglicherweise dieselben anfänglichen Miß- und "Nichtverständnisse" haben wie ursprünglich der Autor. Daher sind diese Fragen, markiert mit "??", sehr nützlich für ein besseres Verständnis des Objekttyps, zumindest sobald sie beantwortet sind, und anhand ihrer kann man oft nachvollziehen, warum genau diese Formulierung und keine andere gewählt wurde. Daher löschen wir die beantworteten Fragen nicht, sondern notieren, eingeschlossen in "!!", die zugehörigen Antworten. Selbstverständlich darf es bei Abschluß des Dokuments (z.B. in Form der Abnahme der Systemspezifikation) keine offenen Fragen mehr geben; Werkzeuge prüfen dies auf syntaktischer Ebene.

Das dritte Querschnittskonzept sind die *optionale Namensteile,* die man an umfassenden eckigen Klammern erkennt, wie Zeile `[04]` zeigt. Ein Merkmal namens *Nummer* ist sicherlich nicht sehr aussagekräftig und daher *kein* gut gewählter Name, denn es wird in sehr vielen Objekttypen vorkommen. Wesentlich besser ist zum Beispiel der Name *BuchungsNummer*, also die Ergänzung des Merkmalnamens um den Objekttypnamen. Bezieht man sich aber von einem anderen Objekttyp auf dieses Merkmal, so erhält man in der Punktnotation sehr leicht einen Namen wie *MeineBuchung.BuchungsNummer*, der ärgerlich lang ist und nicht besonders schön klingt. Durch die Spezifikation *[Buchungs]Nummer* sind in unserer Notation sowohl *MeineBuchung.BuchungsNummer* als auch *MeineBuchung.Nummer* gültige Bezeichner für das Merkmal.

Diese drei Querschnittskonzepte — Markieren, um einer maschinellen Prüfung zugänglich zu machen, die verschiedenen Kommentararten sowie die optionalen Namensteile — sind nicht nur in Objekttypbeschreibungen zulässig, sondern in jedem unserer Dokumente.

Identifizierung von Objekten

Abgeschlossen wird die Spezifikation der Merkmale durch eine Aufzählung der Merkmale bzw. Merkmalskombinationen, die ein Objekt eindeutig identifizieren, also in deren Wert(en) sich je zwei Objekte dieses Objekttyps unterscheiden [**13**]. Implizit wird durch jede Zeile dieser Aufzählung eine Fähigkeit *finden_mit()* spezifiziert, die als Eingabe gerade diese Merkmalskombination hat und höchstens ein Objekt dieses Objekttyps findet.

Spezifikation des Lebenslaufs

Dieser endliche Automat wird in einer Tabelle beschrieben mit den drei Spalten Ausgangszustand, verändernde Fähigkeit und Folgezustand [**14**]. Zwei Zustände hat jeder Lebenslauf, nämlich "präexistent" und "postexistent", also die Zustände des Objekts vor Entstehen bzw. nach Beseitigung. In Anlehnung an die gebräuchlichen Symbole für Geburt und Tod bezeichnen wir sie mit "*" und "+".

Ausgezeichnete zustandsändernde Fähigkeiten

Die Fähigkeiten, die ein Objekt erschaffen, also aus dem Zustand "+" führen, und diejenigen, die das Objekt wieder vernichten, also nach "+" führen, spezifizieren wir nicht wie normale Fähigkeiten, sondern in speziellen Abschnitten "ENTSTEHT_DURCH" [**16**] und "ERLISCHT_DURCH" [**20**].

Nach der *animistischen Auffassung* ist nur das einzelne Objekt in der Lage, sich in den Zustand "+" überzuführen; es begeht also "Selbstmord", d.h. jede vernichtende Fähigkeit ist eine "normale" Objektfähigkeit. Im Gegensatz dazu muß jede erzeugende Fähigkeit eine Typfähigkeit sein, da es ja noch kein Objekt gibt, dem man den Auftrag "schaffe dich" gibt. Man kann nur den Typverwalter beauftragen, ein neues Objekt zur Verfügung zu stellen. An dieser Stelle wollen wir aber eine notationelle Ausnahme zulassen. Statt im einfachsten Fall die Zeilen

```
JEDES               _Objekt

ENTSTEHT_DURCH      anlegen
LIEFERT             Objekt --->     _Objekt
```

zu spezifizieren, so daß man ein neues *_Objekt* nur durch

```
VAR NeuesObjekt : _Objekt
^_Objekt.anlegen(NeuesObjekt)
```

erzeugen kann, ziehen wir folgende Spezifikation vor:

JEDES	_Objekt
ENTSTEHT_DURCH	anlegen

Der Aufruf lautet nun einfach *NeuesObjekt.anlegen()*, d.h. man richtet die Aufforderung, sich anzulegen, direkt an das neu anzulegende Objekt. Diese methodisch abweichende Notation haben wir nur aus pragmatischen Gründen eingeführt.

Wie bei der *Vererbung von Lebensläufen* bereits angedeutet, bewirken die Fähigkeiten, die im Abschnitt "ENTSTEHT_DURCH" spezifiziert werden, **implizit** immer einen Zustandsübergang von "*" nach "existiert" im geerbten Lebenslauf des Vorfahren *_Ding*, d.h. dieser Übergang muß nicht mehr explizit spezifiziert werden. Analog muß der Zustandsübergang von "existiert" nach "+", den jede vernichtende Fähigkeit auslöst, nicht notiert werden.

Spezifikation von Fähigkeiten

bestehen aus bis zu sechs Unterabschnitten. Eingeleitet durch KANN, KANN_MAN oder KANN_SICH wird die Fähigkeit benannt[14] [**21**]. Zwar spiegelt sich das animistische Prinzip im "*kann_sich*" am deutlichsten wider, doch daraus resultierende Formulierungen wie "*kann_sich freigeben()"* klingen zumindest ungewöhnlich.

Es folgen drei Kapitel, in denen die Eingabeparameter (BRAUCHT) und die Ausgabeparameter (LIEFERT) spezifiziert werden ([**22**] oder [**27**]). Eine Besonderheit sind die Eingabeparameter, die von der Fähigkeit direkt in das (meist gleichnamige) Merkmal kopiert werden. Zur Schreiberleichterung haben wir daher den Abschnitt AENDERT eingeführt [**17**], in dem diejenigen Merkmale in Form von Eingabeparametern aufgeführt werden, die von der Fähigkeit nur mit einem neuen Wert belegt werden. Besonders typisch ist diese Art der Belegung für erzeugende Fähigkeiten.

Unsere Fähigkeiten liefern nur über Ausgabeparameter Resultate zurück mit einer Ausnahme: Implizit liefert jede Fähigkeit eine *Rückmeldung*, die eine Aussage darüber macht, wie die Fähigkeit beendet wurde ("ok" oder "nicht ok" sowie weitere Informationen) und wie gegebenenfalls der Nachfolgezustand im Lebenslauf heißt (insbesondere dann, wenn eine Fähigkeit in mehreren Folgezuständen landen kann, siehe *freigeben()*). Dies wird im *Effekt* der Fähigkeit mittels der Anweisung ENDE(*Rückmeldung*, *Folgezustand*) spezifiziert [**19**].

Der nun folgende Abschnitt ANNAHME [**23**] enthält die Vorbedingungen der Fähigkeit, also Zusicherungen, die erfüllt sein müssen, bevor die Fähigkeit überhaupt ausgeführt werden kann. Dieser Abschnitt erleichtert die Spezifikation des eigentliche Fähigkeitsrumpfes erheblich, da er einen Großteil von Fallunterscheidungen vorwegnimmt und so den *Effekt* von Ballast befreit.

---

[14] Ausnahmen sind die erzeugenden und vernichtenden Fähigkeiten, die durch ENTSTEHT_DURCH [**16**] beziehungsweise ERLISCHT_DURCH [**20**] eingeleitet werden (s.o.).

Dieser Effekt nämlich soll möglichst klar machen, wie die Fähigkeit abläuft und welche Spuren ihre Nutzung im System hinterläßt [18]. Wir haben für die Spezifikation des Effekts *keine* fest vorgeschriebene Syntax, denn der Entwickler soll frei sein in der Wahl des am besten geeigneten Sprachmittels. Oft, aber sicherlich nicht immer mögen Pseudocode-Konstrukte angemessen sein, aber auch Entscheidungstabellen, Formeln o.ä., eventuell ergänzt um Beispiele, können den Sachverhalt klar ausdrücken, und oft ist ein gut formulierter freier Text die beste Beschreibung.

Der Typverwalter

Um exemplarübergreifende Merkmale, also die Typmerkmale, zu pflegen und um Typfähigkeiten anbieten zu können, benötigt man eine übergeordnete Einrichtung. Dies ist bei uns der *Typverwalter* [24]. Zu jedem Objekttyp stellen wir uns einen Typverwalter vor, dem wir die Typeigenschaften zuordnen. Die Notation der Merkmale [25] und Fähigkeiten [26] ist identisch zur Notation der "normalen" Eigenschaften. Als Typeigenschaft erkennt man sie ausschließlich daran, daß sie im Dokument nach der Zeile "DER <_Objekttyp>.Verwalter" [24] stehen.

Darüber hinaus gibt es zwei Ausnahmen von Typfähigkeiten, die anders spezifiziert werden: zum einen sind dies alle objekterzeugenden Fähigkeiten [16], zum anderen sind es alle "*finden_mit"*-Fähigkeiten, die implizit nur durch Angabe der Eingabeparameter definiert sind [13].

Spezifikation von Datentypen

Mit der eben beschriebenen Spezifikation der Typeigenschaft wird der Teil des Dokuments abgeschlossen, der den Objekttyp beschreibt. Im Anschluß daran, d.h. im selben Dokument, werden dann die Datentypen spezifiziert, die fachlich zu diesem Objekttyp passen [28]. Denn um eine Dokumenten(typ)inflation zu vermeiden und um alle fachlichen Informationen möglichst dicht zusammenzuhalten, werden Datentypen nicht in eigenen Dokumenten wie "Datentypbeschreibungen" spezifiziert. Bei dem Objekttyp *_Ding*, der ja ganz oben in der Spezialisierungshierarchie steht, sammeln wir die Datentypen, die zwar benutzerdefiniert sind, aber doch so allgemein, daß sie sinnvollerweise nicht einem bestimmten Objekttyp zugeordnet werden können (wie zum Beispiel *_GeldBetrag*). Dahin gehört der hier exemplarisch spezifizierte *_Zeitraum* eigentlich auch, und nicht in die Objekttypspezifikation *_Buchung*.

Datentypdefinitionen sind naturgemäß Konstrukte einer Spezifikation, die schon nahe an der Realisierung sind. Daher sind sie den Werkzeugen besonders zugänglich. Wir können aus ihnen einerseits Datenbankdefinitionen und andererseits Datenstrukturdefinitionen für eine Vielzahl von Programmiersprachen erzeugen. Da diese nicht mehr manuell programmiert werden müssen, ist eine Quelle zahlreicher Programmierfehler bereits weitgehend ausgeschaltet und Änderungen am Datentyp müssen nur noch an einer Stelle vorgenommen werden. Dies verringert den Änderungsaufwand wesentlich, falls ein Fehler in der Typdefinition erst sehr spät auffallen sollte oder sich

(beispielsweise durch eine Gesetzes- oder Normungsänderung) ein Datentyp der Anwendung ändert.

4 Ausblick

Eine Objekttypbeschreibung in der vorliegenden Notation läßt sich zu erheblichen Teilen maschinell auswerten. Wir haben Werkzeuge zur Prüfung der Konsistenz der Spezifikation und setzen sie bereits in der Praxis ein. Zusätzlich läßt sich vieles aus einer Spezifikation dieser Art hinüberretten in die Realisierung und die Testphase. So haben wir auch schon Werkzeuge, die aus den Typdefinitionen für verschiedene Zielsprachen zum Beispiel Codestücke (meist in Form von Copystrecken) generieren, zur Datenbankdefinition oder als Datenstruktur im Programm. Diese Werkzeuge müssen noch vervollständigt werden. Denkbar sind auch Werkzeuge, die kontrollieren, wie die Spezifikation umgesetzt wurde, also ob es zum Beispiel zu jeder Fähigkeit eine Operation gibt.

Zur Zeit definieren wir eine Spracherweiterung, um Benutzerschnittstellen angemessen spezifizieren zu können. Diese soll insbesondere graphische Oberflächen berücksichtigen und maschinell auswertbar sein. Eine methodische Basis zur Spezifikation dieser Dialoge werden die Interaktionsdiagramme aus [Denert91] sein. Wir hoffen, einen großen Teil der Programmierung der Benutzeroberfläche einem Generator überlassen zu können, um so einerseits ein einheitliches "Look and Feel" der Oberfläche zu erreichen und um andererseits den Programmierer zu entlasten von mühseliger "Bit- und Pixelpfriemelei", die gerade bei der Programmierung graphischer Oberflächen einen Großteil der Arbeit ausmacht.

Anschließend werden wir soweit als möglich der Notation ein theoretisches Fundament geben, indem wir eine formale Semantik definieren.

In den nächsten Monaten entstehen verschiedene Fallstudien, die in dieser Notation spezifiziert sind und zum Teil auch realisiert werden in verschiedenen (herkömmlichen und objektorientierten) Sprachen.

Nicht zuletzt soll auch das Metamodell [Hesse92] in dieser Notation niedergelegt werden, wir werden den Einsatz von Hypertextwerkzeugen prüfen, Untersuchungen zum Prototyping anstellen, weitere Dokumentsichten[15] definieren, und manches mehr.

---

[15] Je nach Leserkreis bereiten wir ein Dokument wie eine Objekttypbeschreibung anders auf, um dem Leser möglichst weit entgegenzukommen. Diese Dokumentenaufbereitung nennen wir *Dokumentsicht;* sie umfaßt wesentlich mehr als nur "*pretty printing*" und ist wesentlicher Bestandteil des *Dokumentenorientierten Arbeitens* [Denert93].

Dank

Die hier beschriebenen Konzepte entwickelten sich im Laufe der letzten Jahre aus mehreren Projekten bei sd&m. Die vorgestellte Notation wurde in Zusammenarbeit mit Siemens Nixdorf, KORDOBA, erarbeitet, und wir danken insbesondere N. Höbel für seine kritischen Kommentare und wertvollen Anregungen. Vieles ist uns klarer geworden in Diskussionen mit Kollegen von sd&m, die wir nicht alle nennen können. Stellvertretend seien T. Tensi, R. Fahney, G. Koller, S. Aicher und T. Belzner genannt. G. Scholz hat nicht nur viele Ideen eingebracht, sondern auch eine Werkzeugumgebung zu dieser Methode programmiert.

Literatur

[Coad91] P. Coad, E. Yourdon: *Object-Oriented Analysis*, Prentice-Hall, 1991

[Denert91] E. Denert, J. Siedersleben: *Software-Engineering*, Springer, 1991

[Denert93] E. Denert: *Dokumentenorientierte Softwareentwicklung*, erscheint im Informatik-Spektrum (1993)

[Endres92] A. Endres, J. Uhl: *Objektorientierte Software-Entwicklung. Eine Herausforderung für die Projektführung,* Informatik-Spektrum 15 (1992), S. 255-263

[Hesse92] W. Hesse, G. Merbeth, R.Frölich: *Software-Entwicklung. Vorgehensmodelle, Projektführung, Produktverwaltung*, Oldenburg, 1992

[Löhr93] P. Löhr-Richter: *Methodologie - Methodik - Methode. Was steckt dahinter?*, EMISA-Forum 1, 1993

[Meyer88] B. Meyer: *Object-oriented Software Construction*, Prentice Hall, 1988

[Ritterbach93] B. Ritterbach: *Ein Schlüsselkonzept für das ER-Modell*, EMISA-Forum 1, 1993

[Rumbaugh91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen: *Object-Oriented Modeling and Design*, Prentice-Hall, 1991

[Simonsmeier90] W. Simonsmeier: *Objektorientierte Software-Entwicklung*, Übersetzung von *Object-oriented Software Construction* [Meyer88], Hanser, 1990

Objektorientierte Spezifikation

Konzepte und eine Notation

März 93

Wolfgang Bartsch, Ernst Denert

Technische Universität München

sd&m GmbH München

Technische Universität München
Fakultät für Informatik
Arcisstraße 21
8000 München 2
Telefon (089) 2105 - 2685
Email: Bartsch@Informatik.TU-Muenchen.de