Beschreibung gemeinsamen Verhaltens in einem einheitlichen Objektraum

Heiko Kießling
Uwe Krüger
Universität Karlsruhe, Fakultät für Informatik,
Institut für Betriebs- und Dialogsysteme
D-76128 Karlsruhe, Deutschland
E-Mail: {kiessling, ukrueger}@ira.uka.de

Interner Bericht 13/95

Zusammenfassung

Wir betrachten ein System mit einem einheitlichen Objektraum. In einem einheitlichen Objektraum ist das Objekt das primäre Strukturelement des Systems. In einem solchen System stellt sich in erweiterter Form die Frage nach einem Modell, das gemeinsames Verhalten von Objekten zu beschreiben erlaubt. Wir beschreiben die zu stellenden Forderungen und versuchen dann, diese Frage zu beantworten, indem wir entsprechende Sprachelemente in Form von *Sichten* und *Mustern* beschreiben. Anhand dieser Elemente belegen wir, daß Delegation sowohl auf der Ebene der Objekte als auch als Vererbung auf der Ebene der Klassen lediglich als Grundlage der Realisierung des Modells dienen kann, nicht aber in der Sprache selbst angeboten werden sollte. Das beschriebene Modell findet sich in der Sprache ARISTARCH/L wieder, die das objektbasierte Betriebssystem ARISTARCH/OS unterstützt.

1 Einleitung

Wir betrachten ein System mit einem *einheitlichen* Objektraum. Unter einem Objektraum wird die Menge der Objekte des Systems verstanden. Abstrakt gesehen zeigen Objekte zu einem bestimmten Zeitpunkt ihrer Existenz ein bestimmtes *Verhalten* in der Interaktion mit anderen Objekten. Dieses Verhalten wird vom *Zustand* des Objekts zu diesem Zeitpunkt bestimmt. Nicht vom Zustand bestimmte feste Strukturen des Verhaltens werden als *Verhaltensmuster* bezeichnet. Im allgemeinen wird in objektbasierten Systemen das Verhaltensmuster eines Objekts durch die Realisation seiner Operationen bestimmt, deren Ausführung auf den Werten lokaler Variable beruht. Diese Werte werden durch die Realisation der Operationen als abstrakter Zustand interpretiert. Alternativ kann ein Verhaltensmuster aber auch durch Regeln zum Modifizieren der Realisation von Operationen oder auf eine andere Weise bestimmt sein.

In einem einheitlichen Objektraum ist das Objekt das primäre Strukturelement des Systems. Als Folge daraus besteht der Begriff der Anwendung lediglich latent als eine Menge kooperierender Objekte, manifestiert sich jedoch nicht in einem eigenen Strukturelement, also in einem objektbasierten Programm und einem entsprechenden Prozeß. Die Elemente einer Anwendung, die ihren Zustand beschreiben, entstehen nicht durch Aufruf eines Programms, sondern werden ebenfalls als Objekte dargestellt. Auch Moduln als Strukturelemente eines Programms sind nicht vorhanden.

Aus der Einheitlichkeit des Objektraums folgt direkt die potentielle Langlebigkeit von Objekten. Weil ein Objekt in objektbasierten Systemen immer die Ausführung einer Operation

überdauern kann, in der es entstanden ist, und eine Anwendung in einem einheitlichen Objektraum lediglich die Ausführung einer Operation eines bestimmten Objekts darstellt, kann ein Objekt auch die es erzeugende Anwendung überdauern. Die Langlebigkeit von Objekten erlaubt, vorhandene Information so weit wie möglich bei Anpassungen von Anwendungen zu erhalten, wenn Referenzen auf diese Objekte erhalten bleiben. Dies ist etwa bei der Erstellung von Prototypen von Bedeutung, wenn bereits bei der Erprobung durch den Kunden eingegebene Daten erhalten bleiben sollen.

2 Forderungen

In einem solchen System stellt sich in erweiterter Form die Frage nach einem Modell, das gemeinsames Verhalten unter Objekten zu beschreiben erlaubt. Ausgehend von den obigen Randbedingungen sollen nun anhand von Beispielen die Forderungen beschrieben werden, die an ein geeignetes Objektmodell zu stellen sind.

2.1 Direkte Beschreibung von Objekten

Eine dieser Forderungen ist die nach der *direkten* Beschreibung von Objekten anstelle des Sprachelements der Klasse, die allgemein dann notwendig ist, wenn eine Klasse nur ein Exemplar besitzt. Es handelt sich hierbei weniger um ein programmtechnisches Problem als um eine Entwurfsanomalie. Dies ist in objektbasierten Sprachen oft bei Wertobjekten der Fall, die Zahlen, Zeichen oder Wahrheitswerte darstellen. Diese Wertobjekte besitzen dann unterschiedliche Klassen, weil unterschiedliche Werte sich in der unterschiedlichen Realisation der angebotenen Operationen niederschlagen. Klassen mit nur einem Objekt sind in objektbasierten Sprachen oft auch dann notwendig, wenn in imperativen Sprachen Moduln verwendet werden. Aus diesem Grund fordert Szyperski [29] den Begriff des Moduls in objektbasierten Sprachen. Beispiele sind Moduln mit global oder in einer Anwendung gültigen Konstanten oder Bibliotheken mathematischer Funktionen. Weil aber Anwendungen in einem System der hier betrachteten Art latent sind, tritt der Begriff des Programms und damit des Bindens in den Hintergrund. Als wesentlichen Unterschied zwischen dem Begriff des Moduls und dem des Objekts kann man betrachten, daß ein Modul lediglich zum Zeitpunkt des Übersetzens und Bindens, ein Objekt jedoch auch zur Laufzeit eine Identität besitzt. Die Forderung nach dem Begriff des Moduls wird daher in einem System mit einheitlichem Objektraum zur Forderung nach der direkten Beschreibbarkeit von Objekten.

2.2 Direkte Beschreibung von Aspekten von Objekten

Objekte werden im allgemeinen von unterschiedlichen Arten von Anwendungen des Systems unter unterschiedlichen *Aspekten* betrachtet. Ein eine Person darstellendes Objekt kann etwa von einer Finanzanwendung als Steuerzahler und von einer Firmenbuchhaltung als Arbeitnehmer betrachtet werden. Dasselbe Objekt kann dabei unter verschiedenen Aspekten Unterschiede im Verhalten zeigen. In den meisten objektbasierten Systemen arbeiten Objekte verschiedener Anwendungen in voneinander getrennten Umgebungen. Verschiedene Aspekte eines Objekts im Sinne eines Elements desselben Modells einer realen oder abstrakten Wirklichkeit werden dann in diesen Umgebungen jeweils durch ein eigenes Objekt dargestellt. So stellt in einem solchen System eine Finanzanwendung jede Person lediglich unter ihrem Aspekt als Steuerzahler durch ein Objekt dar, während eine Firmenanwendung dieselbe Person nur als Arbeitnehmer beschreibt. Oft ist aber die Kooperation verschiedener Anwendungen notwendig. So soll

etwa der Steuersatz, ein Attribut des Steuerzahlers, angeglichen werden, wenn das Einkommen, ein Attribut des Arbeitnehmers, sich ändert, obwohl beide von verschiedenen Objekten dargestellt werden. Letztlich besteht also die Notwendigkeit, Objekte in verschiedenen Umgebungen mit gemeinsamem Zustand zu versehen. Dies findet im allgemeinen auf der Grundlage von Dateien, Datenbanken und vergleichbaren Mechanismen statt, auf denen die verschiedenen Anwendungen gemeinsam arbeiten. Die Einheitlichkeit eines Objektraums bietet dagegen die Möglichkeit, Anwendungen wenigstens teilweise direkt auf derselben Objektmenge kooperieren zu lassen. Dasselbe Objekt beinhaltet nun sowohl den Aspekt des Arbeitnehmers als auch den des Steuerzahlers. Die Identität des Objekts bleibt unter unterschiedlichen Aspekten erhalten.

In einem Objektmodell auf der Grundlage eines einheitlichen Objektraums scheint ein eigenes Sprachelement zur Beschreibung von Aspekten aus den folgenden Gründen angemessen.

- Aspekte stellen einen wesentlichen Gesichtspunkt der Beschreibung von Objekten dar und sollten daher im Sinne eines möglichst geringen Abstands zwischen Entwurf und Implementation eines Systems durch die Programmiersprache unterstützt werden.

- Die unter einem bestimmten Aspekt eines Objekts bedeutsamen Operationen sollten mit Elementen der Sprache gekennzeichnet werden. Dies erlaubt in sehr genauer Weise, die Menge der Objekte zu begrenzen, die von Modifikationen dieser Operationen betroffen sind und daher ebenfalls modifiziert werden müssen.

- Dieselben Operationen eines Objekts können entsprechend verschiedener Aspekte unterschiedliches Verhalten zeigen. Es ist daher notwendig, für dieselbe Operation unterschiedliche Realisierungen zu erlauben, die dieses unterschiedliche Verhalten widerspiegeln.

Ein Sprachelement zur direkten Beschreibung von Aspekten soll erlauben, die verschiedenen Aspekte weitgehend getrennt voneinander, aber dennoch innerhalb eines Objekts zu beschreiben.

2.3 Direkte Beschreibung von Rollen von Objekten

Objekte können in einem System oft unterschiedliche *Rollen* mit eigenständigem Verhalten spielen. Spielt ein Objekt unterschiedliche Rollen, so kann die Identität des Objekts im Vergleich zu diesen Rollen in den Hintergrund treten. Obwohl Rollen im allgemeinen zugleich unterschiedliche Aspekte eines Objekts widerspiegeln, unterscheiden sie sich von Aspekten im Sinne des vorangegangenen Absatzes darin, dem sie spielenden Objekt unterschiedliche Identitäten zuzuordnen. Ein Objekt *hat* somit einen bestimmten Aspekt genau einmal, kann aber die gleiche Rolle in unterschiedlichem Zusammenhang mehrfach *spielen*.

Ein eine Person darstellendes Objekt kann etwa in der Buchhaltung unterschiedlicher Firmen als Arbeitnehmer mit eigenen Informationen wie der Personalnummer geführt sein. Gegeben sei konkret ein Objekt, das eine Person darstellt, die bei drei unterschiedlichen Firmen arbeitet. Untersucht man eine Datenstruktur, die als Menge von Personen dient, so kann das betrachtete Objekt sich einmal in dieser Struktur befinden. Betrachtet man hingegen eine Datenstruktur, die als Menge von Arbeitnehmern dient, so kann das Objekt sich durch seine Rollen dreimal in dieser Struktur befinden. Im Gegensatz zu einem Aspekt, der direktes Verhalten eines Objekts beschreibt und entsprechend genau einmal vorhanden ist, können Rollen als separat identifizierbare Elemente auch mehrfach vorhanden sein. Die Modellierung als Aspekt oder als Rolle hängt von den gegebenen Randbedingungen ab. Kann beispielsweise eine Person nur bei einer Firma angestellt sein, ist eine Modellierung als Aspekt denkbar, andernfalls ist eine Modellierung als Rolle notwendig.

Im Gegensatz zu Aspekten sind die Rollen eines Objekts und ihr Verhalten lose an das sie spielende Objekt gebunden. Oft kann das durch eine Rolle beschriebene Verhalten einer Beziehung unter Objekten zugeordnet werden, von denen eines das die Rolle spielende Objekt ist. So kann man etwa das Verhalten der Rolle eines Arbeitnehmers der Beziehung zwischen der Person und dem Arbeitgeber zuordnen. Nicht immer aber kann das Spielen einer Rolle auf eine feste Zielmenge anderer Objekte eingeengt werden. Eine Datei wird etwa in unterschiedlichen Umgebungen arbeiten, dabei aber nicht notwendigerweise in jeder Umgebung von genau einem Objekt benutzt werden. Um den augenblicklichen Zugriffszustand zu bestimmen, wird sie demnach unterschiedliche Rollen spielen, die aber nicht einer Beziehung zu einem bestimmten Objekt zugeordnet werden kann, das sie verwendet. Insgesamt sind in einem Objektmodell auf der Grundlage eines einheitlichen Objektraums mithin Sprachelemente sowohl zur Beschreibung von Aspekten als auch von Rollen angemessen.

2.4 Kategorisierung von Objekten

Sollen Gemeinsamkeiten im Verhalten von Objekten beschrieben werden, so kann man sie in unterschiedliche Kategorien einteilen, die dieses gemeinsame Verhalten ihrer Elemente bestimmen. Dieser Vorgang wird als *Kategorisierung* bezeichnet. Sie findet erstmals statt, wenn ein Objekt in das System eingebracht wird, insbesondere also dann, wenn es erzeugt wird. Eine Kategorie kann aus dem konkreten Verhalten ihrer Elemente definiert sein, aber auch aus einer abstrakten Vorstellung, die ihre Elemente von denen anderer Kategorien unterscheidet. Im ersten Fall kann man von einer *konkreten*, im zweiten von einer *abstrakten Kategorie* sprechen. Bestimmt eine konkrete Kategorie ein fest gegebenes Verhalten, so kann sich das durch eine abstrakte Kategorie bestimmte Verhalten im Laufe der Zeit entsprechend der Anforderungen an die in ihr enthaltenen Objekte ändern. Im folgenden sollen diese beiden Möglichkeiten genau beleuchtet werden.

- Im Falle der *konkreten Kategorisierung* wird ein Objekt auf der Grundlage seines konkreten Verhaltens der entsprechenden konkreten Kategorie zugeordnet. Dies soll im folgenden am Beispiel der bereits von Lieberman [12] vorgestellten Elefanten Clyde und Fred erläutert werden.

  Anfangs sei Clyde der einzige Elefant in einem Zoo. In einem Besucherinformationssystem wird Clyde durch ein Objekt dargestellt. Eines Tages kommt der Elefant Fred hinzu. Wir bemerken, daß Fred in seinem modellierten Verhalten Clyde entspricht und weisen ihn daher derselben konkreten Kategorie zu. Diese Kategorie ist definiert durch das konkrete Verhalten des Objekts, das Clyde darstellt, zu dem Zeitpunkt der Erzeugung des Objekts, das Fred darstellt. Ändert sich nun allerdings ein Teil des Verhaltens eines der Elefanten, verliert beispielsweise Clyde durch einen Unfall eines seiner Beine, so überträgt sich dies nicht auf Fred. Clyde wird durch den Unfall implizit einer anderen konkreten Kategorie als Fred zugeordnet. Jede Modifikation des Verhaltens eines Objekts bewirkt damit eine konkrete *Rekategorisierung*. Die Einordnung von Fred in dieselbe konkrete Kategorie wie Clyde galt nur zum Zeitpunkt der Kategorisierung. Weil die konkrete Kategorie eines Objekts durch sein Verhalten definiert ist, ändert sich die konkrete Kategorie des Objekts bei der Modifikation seines Verhaltens.

  Die an einer konkreten Kategorisierung orientierte Erzeugung von Objekten, wie beispielsweise die des Objekts Fred, geschieht auf der Grundlage eines oder mehrerer bereits vorhandener Objekte, deren Verhalten als Vorlage dienen soll. Das erzeugte Objekt kann aber unmittelbar danach getrennt von den als Vorlage dienenden Objekten modifiziert

und damit einer anderen konkreten Kategorie zugeordnet werden. Die konkrete Kategorisierung erhält damit *momentanen* Charakter.

- Der *abstrakten Kategorisierung* geht immer eine Abstraktionsleistung der Person voraus, die die abstrakte Kategorie definiert, indem sie sie von anderen abstrakten Kategorien unterscheidet. Die entstandene Kategorie ist damit unabhängig von einem zu einem bestimmten Zeitpunkt vorliegenden Verhalten der in ihr enthaltenen Objekte. Bestimmt sich demnach die konkrete Kategorie aus dem Verhalten ihrer Elemente, so ergibt sich umgekehrt das Verhalten der Elemente einer abstrakten Kategorie aus der hinter der Kategorie stehenden abstrakten Vorstellung.

  Nehmen wir beispielsweise an, daß im Zoo neben den Elefanten noch weitere Tierarten vertreten sind. Die Besucher sollen zu jedem Tier über einen Bildschirm Informationen über seinen Lebensraum und seine Lebensgewohnheiten abrufen können. Entsprechend werden die Tiere nach ihrer Art abstrakt kategorisiert. Diese Kategorisierung bestimmt nun für jedes Tier und insbesondere für die Elefanten entsprechend seiner abstrakten Kategorie sein Verhalten. Ändert sich die Vorstellung einer Tierart, so ändert sich das durch die abstrakte Kategorie der Tierart vorgegebene Verhalten. Obwohl bei einer Modifikation der Beschreibung des Lebensraums der Elefanten Objekte modifiziert werden, bleibt ihre Zuordnung zu der abstrakten Kategorie der Elefanten erhalten. Die abstrakte Kategorisierung hat anders als die konkrete Kategorisierung *permanenten* Charakter, der sich darin ausdrückt, daß die Modifikation des Verhaltens von Objekten, das durch ihre abstrakte Kategorie bestimmt ist, zugleich für alle Objekte in dieser Kategorie stattfindet.

  Neben dem permanenten Charakter zeichnet sich eine abstrakte Kategorisierung im Gegensatz zur konkreten Kategorisierung im allgemeinen dadurch aus, daß sie nicht das gesamte modellierte Verhalten der kategorisierten Objekte festschreibt. Teile des Verhaltens sind entweder nicht oder lediglich durch ihre Struktur im Sinne eines Verhaltensmusters bestimmt. Es wird darum zwischen für die der abstrakten Kategorie zugrundeliegenden Vorstellung *wesentlichen* oder *charakteristischen* und *unwesentlichen* oder *nebensächlichen* Verhaltensweisen unterschieden. So stellt etwa die Tatsache, daß ein Elefant einen Rüssel mit einer bestimmten Länge besitzt, ein wesentliches Verhaltensmuster oder genauer eine wesentliche Eigenschaft dar, während die individuelle Länge von Elefant zu Elefant verschieden sein kann und damit für die abstrakte Kategorie Elefant unwesentlich ist.

  Insgesamt lassen sich demnach drei unterschiedliche Stufen der abstrakten Kategorisierung unterscheiden.

  - *Kategorisierung von Verhalten.* Klassenbasierte Systeme erlauben Klassen im allgemeinen, einen eigenen Zustand zu besitzen. Dieser Zustand ist des impliziten Zugriffs wegen letztlich allen Exemplaren einer Klasse gemeinsam. Die Exemplare werden demnach neben den durch die Klasse gegebenen Operationen durch den Zustand der Klasse kategorisiert. Damit erstreckt sich die Kategorisierung auf einen Teil des Verhaltens, der zu jedem Zeitpunkt allen Exemplaren gemein ist. Als Beispiel kann die Beschreibung des Lebensraums einer durch eine Kategorie dargestellten Tierart dienen.
  - *Kategorisierung von Verhaltensmustern.* In klassenbasierten Sprachen geben Klassen im allgemeinen ihren Exemplaren die Realisation der Operationen und die Menge der lokalen Variablen vor. Die abstrakte Kategorisierung bezieht sich auf durch diese Realisation festgelegte Verhaltensmuster, etwa auf die Eigenschaft von Elefanten,

einen Rüssel mit einer individuellen Länge zu haben. Die Länge des Rüssels eines bestimmten Elefanten ist nicht durch die abstrakte Kategorie festgelegt, wohl aber die Eigenschaft, einen Rüssel zu besitzen.

– *Keine Kategorisierung.* Schließlich ist es möglich, daß verschiedene Objekte einer abstrakten Kategorie Verhalten besitzen, das nicht einmal einem gemeinsamen Muster folgt. Dieses Verhalten bleibt bei der abstrakten Kategorisierung unberücksichtigt, ist also für die der Kategorie zugrundeliegende Vorstellung unwesentlich. Nehmen wir etwa an, einer der Elefanten in unserem Zoo zeigt die bisher nicht dagewesene Fähigkeit, auf seinen Hinterbeinen laufen zu können. Dies mag der erste Schritt der Evolution hin zu aufrecht gehenden Elefanten sein, im Augenblick aber ist dieser Elefant einmalig. Seine Einteilung in dieselbe Kategorie wie seine Artgenossen ist daher zwar sinnvoll, aber dennoch unterscheiden sich Teile seines Verhaltens grundlegend von allen anderen. In einem solchen unwesentlichen Verhalten kann ein Objekt beliebig modifiziert werden, ohne seine Zugehörigkeit zu seiner abstrakten Kategorie zu verlieren.

Nicht durch eine abstrakte Kategorie erfaßtes Verhalten kann wiederum zur Bildung weiterer konkreter Kategorien verwendet werden. Bei der Erzeugung von Objekten entsprechend einer solchen konkreten Kategorie bleibt jedoch die Zuordnung zur zugrundeliegenden abstrakten Kategorie erhalten. Der nicht durch die abstrakte Kategorie kategorisierte Teil des Verhaltens der Objekte ist wieder getrennt modifizierbar.

Die abstrakte Kategorisierung von Verhalten kann zu unterschiedlichen Kategorien führen, die sich in diesem Verhalten unterscheiden, denen für dieses Verhalten jedoch ein gemeinsames Verhaltensmuster zugrundeliegt. Damit ergibt sich eine weitere wichtige Forderung an eine Sprache in einem System mit einheitlichem Objektraum. Die Kategorien müssen selbst wieder entsprechend des dem von ihnen kategorisierten Verhalten zugrundeliegenden Verhaltensmusters kategorisiert werden können; das entspricht der Bildung von Kategorien *höherer Ordnung*. Dies soll am Beispiel einer Zeichenkettenklasse verdeutlicht werden. Um die Darstellung von Zeichenketten und ihren Vergleich effizient zu gestalten, wird oft eine Tabelle angelegt, in der jede auftretende Zeichenfolge einen Eintrag besitzt. Eine Zeichenkette besitzt einen Index in diese Tabelle, wobei Objekte mit derselben Zeichenfolge denselben Index enthalten. Die Tabelle kann somit als gemeinsamer Zustand aller Zeichenketten aufgefaßt werden und findet ihren natürlichen Platz in der Klasse der Zeichenketten, die dadurch neben ihrem Verhaltensmuster durch die Benutzung dieser Tabelle kategorisiert sind. Ist die Herausgabe von Zeichenketten aus einem bestimmten Umfeld, beispielsweise einer Anwendung, nicht notwendig, so ist es vor allem aus Effizienzgründen sinnvoll, innerhalb dieses Umfeldes eine eigene Zeichenfolgentabelle zu verwenden. Dies tritt umso deutlicher hervor, wenn man einen Mehrbenutzerobjektraum annimmt. Ein solcher Objektraum erlaubt mehreren Benutzern, die durch Stellvertreterobjekte repräsentiert werden, über gemeinsame Objekte die Kooperation miteinander. In diesem Fall kann es nicht nur ineffizient, sondern regelrecht falsch sein, allen Benutzern dieselbe Zeichenkettenklasse und damit dieselbe Zeichenfolgentabelle zuzuordnen, da je nach angebotenem Funktionsumfang der Zeichenketten ein Benutzer herausfinden kann, welche Zeichenketten ein anderer Benutzer definiert hat. Es sind mehrere Zeichenkettenklassen notwendig, auf deren Benutzung getrennt Rechte vergeben werden können, die jedoch gemeinsam beschrieben werden sollen. In herkömmlichen Begriffen sind also Klassen *höherer Ordnung* notwendig, also Klassen zur Kategorisierung von Klassen, die ihren Exemplaren neben demselben Verhaltensmuster auch den durch die Kategorie kategorisierten Zustand vorgeben, sich aber in diesem unterscheiden können.

2.5 Umsetzung

Insgesamt sind Forderungen an eine Sprache in einem System mit einheitlichem Objektraum folgendermaßen zusammenzufassen. Die Sprache muß Elemente zur direkten Beschreibung von Objekten und ihren unterschiedlichen Aspekten und Rollen anbieten. Sie muß ferner die Unterscheidung zwischen der konkreten Kategorisierung des Verhaltens von Objekten und der abstrakten Kategorisierung von Teilen des Verhaltens erlauben, um insbesondere Modifikationen sowohl auf einzelne Objekte als auch auf Mengen von Objekten beziehen zu können. Schließlich muß sie die Beschreibung gleichartiger Objektmengen unterstützen, in denen Untermengen gemeinsamen Zustand besitzen.

Einige der hier angegebenen Forderungen scheinen durch Delegation sowohl auf der Ebene der Objekte als auch durch Vererbung auf der Ebene der Klassen erfüllt zu sein. So argumentiert Lieberman für Delegation als Mechanismus zur abstrakten Kategorisierung, während die in breitem Gebrauch befindlichen objektorientierten Sprachen Klassen und Vererbung verwenden. In den folgenden Abschnitten wollen wir nun ein Modell zur Beschreibung von Gemeinsamkeiten im Objektverhalten vorstellen, das die aufgestellten Forderungen in einer nach unserer Ansicht hinreichenden Weise erfüllt. Wir werden dabei belegen, daß Delegation sowohl auf der Ebene der Objekte als auch als Vererbung auf der Ebene der Klassen lediglich als Grundlage der Realisierung des Modells dienen kann, nicht aber in der Sprache selbst angeboten werden sollte. Die Implementation der Sprachelemente mit Ausnahme des Bezugs zu Delegation ist nicht Gegenstand dieses Berichts.

In Abschnitt 3 werden wir zunächst einige Begriffe klären, worauf dann in Abschnitt 4 unser Ansatz zur Beschreibung unterschiedlicher Aspekte eines Objekts folgt. In Abschnitt 5 beschäftigen wir uns dann mit dem Problem der Kategorisierung und der Rollen. Beide Abschnitte schließen mit einer Kritik an der Idee der Delegation. Das beschriebene Modell findet sich in der Sprache ARISTARCH/L wieder, die das objektbasierte Betriebssystem ARISTARCH/OS unterstützt. Beispiele werden daher anhand dieser Sprache dargestellt, aber zugleich im Text erläutert.

3 Objekte

In diesem Abschnitt soll in knapper Form die diesem Bericht zugrundeliegende Terminologie erläutert werden. Es wird im folgenden immer wieder notwendig sein, streng zwischen der *Definition* und der *Implementation* der Komponenten eines Objekts zu unterscheiden. Zu Anfang soll es ausreichen, als Komponenten eines Objekts Attribute und Befehle zu betrachten. Ein Befehl beschreibt einen durch das Objekt angebotenen Dienst. Attribute bestehen aus einem Befehl zum Lesen und einem zum Schreiben des Attributs. Dies sind zugleich die Begriffe für die Definition dieser Komponenten. Ferner beschreibt jedes Objekt eine Implementation, die festlegt, wie Attribute und Befehle realisiert werden. Jeder Befehl kann durch eine Methode implementiert werden. Jedes Attribut kann entsprechend durch eine Methode zum Lesen und eine zum Schreiben, oder alternativ durch eine Variable implementiert werden, die implizit eine Methode zum Lesen und eine zum Schreiben definiert. Dies erlaubt die Anpassung von Attributimplementationen auf dieselbe Weise wie etwa in TRELLIS/OWL von Schaffert und Cooper [23]. Definition und Implementation eines Objekts werden in ARISTARCH/L durch einen Objektausdruck beschrieben, den man als einen Ausdruck auffassen kann, der zu einer konstanten Referenz auf das beschriebene Objekt ausgewertet wird. Jeder Objektausdruck hat hierzu einen `IMPLEMENTATION`-Abschnitt, der wiederum in einen `ATTRIBUTES`-Teil zur Definition und Implementation der Attribute und einen `COMMANDS`-Teil zur Definition und Implementation der Befehle gegliedert ist. Damit ist die direkte Beschreibung von Objekten möglich, so daß die

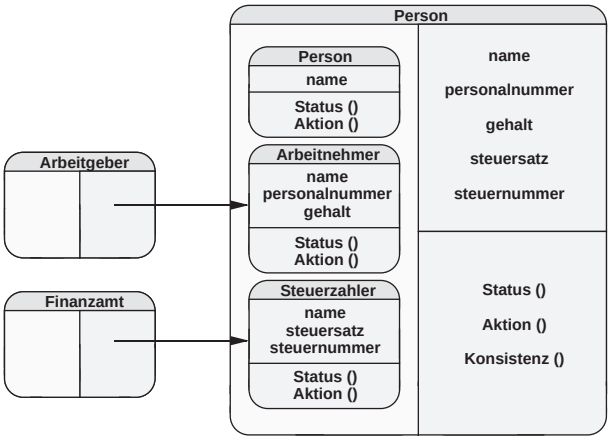

Bild 4.1: Aspekte einer Person und ihre Nutzer

erste Forderung erfüllt ist.

4 Sichten

4.1 Grundlagen

Objekte werden von unterschiedlichen Arten von Anwendungen des Systems unter unterschiedlichen Aspekten betrachtet. Beispielsweise kann ein eine bestimmte Person darstellendes Objekt von einer Finanzanwendung als Steuerzahler und von einer Firmenbuchhaltung als Arbeitnehmer betrachtet werden. Bild 4.1 zeigt eine derartige Konstellation. In ARISTARCH/L besitzt nun jedes Objekt eine oder mehrere *Sichten*. Sichten beschreiben die verschiedenen Aspekte eines Objekts durch Angabe einer logisch in sich abgeschlossenen Komponentenmenge, der der Sicht zugeordneten *Schnittstelle*. Die Schnittstellen der verschiedenen Sichten sind im `INTERFACES`-Abschnitt des Objekts enthalten.

Neben der Darstellung verschiedener Aspekte eines Objekts dienen die Schnittstellen der Sichten dem Geheimnisprinzip. Alle nicht in der Schnittstelle einer Sicht des Objekts angegebenen Komponenten sind anderen Objekten unbekannt. Die durch keine Sicht bekanntgemachten Komponenten des Objekts können beliebig gestrichen oder umbenannt werden, ohne daß andere Objekte davon berührt sind.

Ein Objekt wird immer unter einer der zur Verfügung stehenden Sichten betrachtet. Dazu beschreibt jede Referenz zusätzlich zur Objektidentität auch die Identität der aktuellen Sicht, unter der das Objekt über die jeweilige Referenz gesehen werden soll. Es ist jedoch möglich, aus einer Referenz Referenzen auf andere Sichten desselben Objekts zu erhalten.

Beispiel 4.1 beschreibt im linken Kodefragment die programmiersprachliche Umsetzung dreier derartiger Sichten auf eine `person`. Eine Sicht `Person` beschränkt sich auf die allgemeinen personenbezogenen Daten wie den Namen (Zeilen 3 bis 9), die Sicht `Arbeitnehmer` umfaßt daneben auch die Personalnummer und das Gehalt (Zeilen 11 bis 19), während die Sicht als `Steuerzahler` (Zeilen 21 bis 29) stattdessen den Steuersatz und die Steuernummer enthält[1]. Das Schlüsselwort `IMMUTABLE` vor einer Attribut-Deklaration in einer Sicht schränkt die Nutzbarkeit des entsprechenden Attributs auf das Auslesen des Wertes ein. Beschreibt ein Objektausdruck, wie in diesem Beispiel, mehrere Sichten, muß mittels einer `AS`-Klausel auch die initiale

[1] *Ableitung* von Sichten erlaubt eine einfachere Beschreibung derartiger Konstellationen, wird jedoch erst in Abschnitt 4.5 behandelt.

```
OBJECT
  INTERFACES
    VIEW Person
      ATTRIBUTES
        IMMUTABLE name: tString
      COMMANDS
        Status ();
        Aktion ()
    END VIEW Person;

    VIEW Arbeitnehmer
      ATTRIBUTES
        IMMUTABLE name: tString;
        personalnummer: INTEGER;
        gehalt: INTEGER
      COMMANDS
        Status ();
        Aktion ()
    END VIEW Arbeitnehmer;

    VIEW Steuerzahler
      ATTRIBUTES
        IMMUTABLE name: tString;
        steuernummer: INTEGER;
        IMMUTABLE steuersatz: INTEGER
      COMMANDS
        Status ();
        Aktion ()
    END VIEW Steuerzahler;
  ...
END OBJECT AS Person -> person
```

```
1  OBJECT
       ...
     IMPLEMENTATION
       COMMANDS
         Steuererklaerung (person: tPerson)
           BY METHOD
             DECLARE
               ATTRIBUTES
                 s: tSteuerzahler BY VARIABLE;
             ...
             DO VIEW person AS Steuerzahler -> s;
             ... s.steuersatz ...
           END Steuererklaerung;
     ...
15 END OBJECT -> finanzamt
```

Beispiel 4.1.: Aspekte einer Person

Sicht bestimmt werden, unter der das Objekt betrachtet werden soll (Zeile 31). Die Anwendung `Steuererklaerung` in einem Finanzamt, die eine Referenz auf ein Objekt unter der allgemeinen Sicht `Person` erhält, kann die Person dann unter dem ihr genehmen Aspekt, nämlich dem des Steuerzahlers betrachten, indem über die Wandeloperation `VIEW` eine Referenz beschafft wird, die auf die gewünschte Sicht desselben Objekts verweist (Zeile 11 des rechten Kodefragments). Diese Referenz erlaubt dann den Zugriff auf die in der Sicht `Steuerzahler` angegebene Schnittstelle und damit auf das Attribut `steuersatz`. Man beachte, daß jede Sicht als Beschreibung eines Aspekts eines Objekts genau einmal vorhanden ist.

Jede Komponente kann verschiedene Implementationsversionen besitzen, um die verschiedenen Aspekte des Objekts zu berücksichtigen. So sind jedes Attribut und jeder Befehl durch verschiedene Variablen und Methoden implementierbar. Beispielsweise soll ein Befehl `Status` Informationen über eine `person` ausgeben. Diese Informationen werden für einen Steuerzahler andere sein als für einen Arbeitnehmer. Der Programmierer wird daher verschiedene Versionen der Methode erstellen, die den Befehl `Status` implementiert.

4.2 Auswahl von Implementationsversionen

Um die Implementationsversionen nun auch unter den entsprechenden Sichten ausführen zu können, muß jede Sicht eine *Abbildung* enthalten, die jeder Komponente eine bestimmte Version zuordnet. Jede Version muß dabei in der Abbildung wenigstens einer Sicht zugeordnet werden. Da verschiedene Sichten dieselbe Version verwenden können, ist es nicht sinnvoll, diese Abbildung durch syntaktische Schachtelung der Versionen in den Sichten zu beschreiben. In ARISTARCH/L erhalten die verschiedenen Versionen stattdessen Namen, auf die die Abbildung sich beziehen kann. Diese Versionsnamen sind nur in bezug auf die Komponente, also nur gemeinsam mit dem Namen dieser Komponente eindeutig. Dahinter steckt eine grundlegende Idee

in ARISTARCH/L von *Namen als Spezifikation* der Programmkomponenten, die sie bezeichnen. Diese Idee liegt auch dem Typsystem von ARISTARCH/L zugrunde, das in dieser Hinsicht dem von EMERALD [3] vergleichbar ist, aber nicht Gegenstand dieses Berichts sein soll.

Während im Beispiel 4.1 die Sichten lediglich unterschiedliche Teilmengen der Objektkomponenten einer Person sichtbar machen, zeigt Beispiel 4.2 anhand des Befehls `Status` die Verwendung unterschiedlicher sichtenspezifischer Versionen. Im Implementationsabschnitt des Objekts werden die drei Versionen für den allgemeinen Personenaspekt, den Arbeitnehmeraspekt und den Steuerzahleraspekt vereinbart (Zeilen 21 bis 29), welche die dem jeweiligen Aspekt entsprechenden Informationen ausgeben. In der Abbildung, eingeleitet durch das Schlüsselwort `MAPPINGS`, werden den Sichten dann die entsprechenden Implementationsversionen zugeordnet (Zeilen 4 bis 13). Komponenten wie zum Beispiel `Status` sind mit dem Schlüsselwort `ASSERTED` gekennzeichnet (Zeilen 16, 17, 20 und 31). Diese Kennzeichnung müssen alle Komponenten tragen, die an einer Schnittstelle angeboten werden sollen. Sie hat den Vorteil, daß nicht durch Sichten angebotene Komponenten direkt in der Implementation erkennbar sind.

Das Objekt selbst, also seine Implementation, kann alle seine Komponenten verwenden. Es greift demnach nicht unter einer bestimmten Sicht auf die eigenen Komponenten zu. Damit stellt sich die Frage nach der Auswahl der Implementationsversion einer durch das Objekt selbst benutzten Komponente. Nehmen wir hierzu beispielsweise an, der Befehl `Status`, der Information über eine Person ausgibt, soll nun durch `person` selbst zum Zweck der Fehlersuche verwendet werden, wann immer durch ein anderes Objekt ein Befehl erteilt wird. In diesem Fall reicht es aus, entsprechend der verschiedenen Sichten des Objekts verschiedene Versionen jeder Methode zu erstellen, die sich lediglich im Aufruf eines anderen Ausgabebefehls unterscheiden. Dies ist allerdings ein sehr aufwendiger Weg. Er ist aber auch unzureichend, wenn die Befehle manchmal durch `person` selbst aufgerufen werden. Im allgemeinen sollte vielmehr die Sicht die Implementationsversion einer Komponente bestimmen, unter der durch ein anderes Objekt die Komponente verwendet wurde, die letztlich direkt oder indirekt auf die zuzuordnende Komponente zugreift. Die Abbildung einer Sicht hat daher *allen* Komponenten des Objekts eine Version zuzuordnen, nicht nur den in der Schnittstelle der Sicht angegebenen. Existiert von einer Komponente nur eine Version, so muß sie entsprechend in der Abbildung einer Sicht nicht explizit angegeben werden.

In Beispiel 4.2 ist der Befehl `Aktion` über alle Sichten zugänglich (Zeile 8, 18 und 28 des linken Kodefragments in Beispiel 4.1), besitzt jedoch nur eine Implementationsversion (Zeilen 31 bis 33). Daher kann auf einen Eintrag im Abbildungsabschnitt verzichtet werden. Diese Implementation verwendet den Befehl `Status`, um Informationen auszugeben. Die zur Ausführung gewählte Implementation beim Aufruf des `Status`-Befehls (Zeile 32) ist dabei von der Abbildung der Sicht abhängig, über die der Befehl `Aktion` erteilt wurde. Ein Aufruf über die `Arbeitnehmer`-Sicht führt damit zur Ausführung der `ArbeitnehmerVersion` des `Status`-Befehls, während ein Aufruf über die `Steuerzahler`-Sicht die Ausführung der `SteuerzahlerVersion` zur Folge hat.

4.3 Erweitern von Versionen

Oft kann man eine Implementationsversion einer Komponente als Erweiterung einer anderen betrachten. In Beispiel 4.2 etwa geben sowohl die `ArbeitnehmerVersion` als auch die `SteuerzahlerVersion` von `Status` die Personendaten aus und können damit als Erweiterungen der `GrundVersion` aufgefaßt werden. Diese Beziehung kann durch ein Sprachelement programmtechnisch ausgenutzt werden, das Versionen in ihrer Eigenschaft als Erweiterungen einer Grundversion beschreibt. Es reicht hierbei aus, ein solches Element für Methoden anzugeben, denn Attribute werden als Paare von Befehlen verstanden, die durch Paare von Methoden

```
1 OBJECT
    ...
    IMPLEMENTATION
      MAPPINGS
5       VIEW Person ...
          Status BY GrundVersion; ...
        END VIEW Person;
        VIEW Arbeitnehmer ...
          Status BY ArbeitnehmerVersion; ...
10      END VIEW Arbeitnehmer;
        VIEW Steuerzahler ...
          Status BY SteuerzahlerVersion; ...
        END VIEW Steuerzahler

15    ATTRIBUTES
        ASSERTED name: tString BY VARIABLE;
        ASSERTED personalnummer: INTEGER BY VARIABLE;
        ...
      COMMANDS
20      ASSERTED Status () BY
          VERSION GrundVersion BY METHOD
            Ausgabe allgemeiner Personendaten
          END VERSION GrundVersion,
          VERSION ArbeitnehmerVersion BY METHOD
25          Ausgabe allgemeiner Personendaten, Ausgabe Arbeitnehmerdaten
          END VERSION ArbeitnehmerVersion,
          VERSION SteuerzahlerVersion BY METHOD
            Ausgabe allgemeiner Personendaten, Ausgabe Steuerzahlerdaten
          END VERSION SteuerzahlerVersion
30      END Status;
        ASSERTED Aktion () BY METHOD
            ... Status (); ...
        END Aktion;
        ...
35  END OBJECT AS Person -> person
```

Beispiel 4.2.: Sichten und Versionen

implementiert werden.

Im Sinne eines entsprechenden Elements stellt jede Methodenversion einfach eine Folge von Anweisungen dar, die gemeinsame Eingabe- und Ausgabeparameter, aber auch lokale Definitionen besitzt und mit Anweisungsfolgen anderer Versionen zu einer Gesamtversion zusammengesetzt werden kann. Die durch die Erweiterung entstehende Version ist das Ergebnis einer Verknüpfung der Anweisungsfolge der Grundversion und der der Erweiterung. Der Seiteneffekt auf den Objektzustand besteht daher aus dem Beitrag der Grundversion und dem der Erweiterung. Um die Reihenfolge der Beiträge beschreiben zu können, muß der Punkt bestimmbar sein, an dem in bezug auf die Anweisungsfolge der Erweiterung die Anweisungsfolge der Grundversion ausgeführt wird. Ferner hat das Sprachelement Sorge zu tragen, daß die Grundversion immer genau einmal ausgeführt wird, also in jedem Pfad der Erweiterung unabhängig von Objektzustand und Argumenten. In ARISTARCH/L erreichen wir all dies, indem syntaktisch um den Punkt der Grundversion zwei Anweisungsfolgen gruppiert sind. Alle Pfade der ersten Anweisungsfolge der Erweiterung konvergieren somit in der Grundversion, um dann in der zweiten Anweisungsfolge der Erweiterung wieder zu divergieren. Die Angabe des Punkts der Grundversion durch das Schlüsselwort `BASE METHOD` ist zwingend, denn dem Programmierer einer Version ist nicht von vornherein bekannt, ob diese Version einmal als Erweiterung einer anderen verwendet wird.

Diese Vorgehensweise ist in Beispiel 4.3 anhand des Befehls `Status` dargestellt. Während die `Grundversion` die Ausgabe der allgemeinen personenbezogen Daten enthält (Zeilen 2 bis 5), erweitert die `ArbeitnehmerVersion` durch eine `EXTENDING`-Klausel diese Version und beschränkt sich jetzt auf die Ausgabe der arbeitnehmerbezogenen Daten, wie das Gehalt und die Perso-

```
1 ASSERTED Status () BY
    VERSION GrundVersion BY METHOD
      BASE METHOD
      DO Ausgabe allgemeiner Personendaten
5   END VERSION GrundVersion,
    VERSION ArbeitnehmerVersion EXTENDING GrundVersion BY METHOD
      BASE METHOD
      DO Ausgabe Arbeitnehmerdaten
    END VERSION ArbeitnehmerVersion,
10  VERSION SteuerzahlerVersion EXTENDING GrundVersion BY METHOD
      BASE METHOD
      DO Ausgabe Steuerzahlerdaten
    END VERSION SteuerzahlerVersion
14 END Status;
```

Beispiel 4.3.: Erweitern von Versionen

nalnummer. Der Methodenrumpf teilt sich in zwei Anweisungsfolgen, die die Stelle der Grundversionen, falls vorhanden, einschließen. Im Beispiel ist die erste Anweisungsfolge leer. An der Stelle des Schlüsselworts `BASE METHOD` werden durch die `GrundVersion` die allgemeinen personenbezogenen Daten ausgegeben, gefolgt von der Ausgabe der arbeitnehmerbezogenen Daten, wie sie in der zweiten Anweisungsfolge der `ArbeitnehmerVersion` festgelegt ist (Zeilen 6 bis 9). Analog wird für die `SteuerzahlerVersion` vorgegangen (Zeilen 10 bis 13). Es kann demnach die gleiche Grundversion genutzt werden, um verschiedene Versionen mit unterschiedlichen Erweiterungen zu realisieren, ohne den entsprechenden Kode vervielfältigen zu müssen.

Ebenso wie der Seiteneffekt auf den Objektzustand bestehen auch die Resultate eines Befehls aus dem Beitrag der Grundversion und dem Beitrag der Erweiterung. Die Erweiterung beschreibt dazu die Resultate der entstehenden Methodenversion als Ergebnis von Verknüpfungen des Beitrags der Grundversion mit dem eigenen Beitrag. In ARISTARCH/L werden die Resultate durch Zuweisung an die Ausgabeparameter bestimmt, die als allen Versionen gemeinsame Attribute betrachtet werden. Dies erlaubt Resultatberechnungen sowohl vor den Anweisungen der Grundversion in der ersten Anweisungsfolge der Erweiterung als auch nach den Anweisungen der Grundversion in der zweiten Anweisungsfolge. Durch aufeinanderfolgende Modifikationen der Ausgabeparameter werden die Resultate schrittweise durch die verschiedenen Methodenversionen entwickelt. Die Ausgabeparameter sind daher zu Anfang mit Werten zu initialisieren, die sich neutral in bezug auf die Verknüpfungen verhalten, durch die die Versionen ihre eigenen Beiträge einbringen. Da durch eine Initialisierung in der ersten Anweisungsfolge der Erweiterung Beiträge möglicherweise vorhandener zusätzlicher Erweiterungen verlorengehen, ist ein dedizierter Mechanismus notwendig, der die Initialisierung unmittelbar nach dem Aufruf des entsprechenden Befehls, aber vor der ersten Anweisung einer der beteiligten Versionen durchführt. In ARISTARCH/L wird dazu in der Liste der Ausgabeparameter der Implementation eines Befehls für jeden Parameter ein Initialwert angegeben.

In Beispiel 4.4 wollen wir einen Befehl zum Prüfen der Zustandskonsistenz des Objekts `person` beschreiben (Zeilen 17 bis 35), der sich auf den der Sicht entsprechenden Teil beschränkt. So prüft etwa die `ArbeitnehmerVersion` nach der Ausführung der `GrundVersion` die entsprechende lokale Konsistenz und legt das Ergebnis in `lokalKonsistent` ab (Zeile 31). Dieses lokale Ergebnis wird dann in der darauffolgenden Zeile mittels der Operation `AND` mit dem Beitrag der `GrundVersion` verknüpft. Diese arbeitet wie die `ArbeitnehmerVersion`, besitzt aber selbst keine Grundversion. Das Schlüsselwort `BASE METHOD` ist jedoch anzugeben, es steht in diesem Fall für eine leere Anweisungsfolge. Die `GrundVersion` verknüpft ihren Beitrag daher mit dem in der Liste der Ausgabeparameter angegebenen Initialwert `TRUE` (Zeile 17), der das neutrale Element der Operation `AND` darstellt.

```
OBJECT
  ...
  IMPLEMENTATION
    MAPPINGS
      VIEW Person ...
        Konsistenz BY GrundVersion
      END VIEW Person;
      VIEW Arbeitnehmer ...
        Konsistenz BY ArbeitnehmerVersion
      END VIEW Arbeitnehmer;
      VIEW Steuerzahler ...
        Konsistenz BY SteuerzahlerVersion
      END VIEW Steuerzahler

    COMMANDS
      ...
      Konsistenz () -> (TRUE -> konsistent: BOOLEAN) BY
        VERSION GrundVersion BY METHOD
          DECLARE
            ATTRIBUTES
              lokalKonsistent: BOOLEAN
          BASE METHOD
          DO Lokale Konsistenzprüfung für Person -> lokalKonsistent;
             lokalKonsistent AND konsistent -> konsistent
        END VERSION GrundVersion,
        VERSION ArbeitnehmerVersion EXTENDING GrundVersion BY METHOD
          DECLARE
            ATTRIBUTES
              lokalKonsistent: BOOLEAN
          BASE METHOD
          DO Lokale Konsistenzprüfung für Arbeitnehmer -> lokalKonsistent;
             lokalKonsistent AND konsistent -> konsistent
        END VERSION ArbeitnehmerVersion,
        VERSION SteuerzahlerVersion EXTENDING GrundVersion BY METHOD ...
      END Konsistenz
END OBJECT AS Person -> person
```

Beispiel 4.4.: Versionen und Resultate

Um auch das Erweitern von Attributen zu beschreiben, die durch Variablen implementiert sind, erinnern wir uns, daß eine Variable implizit eine Methodenversion zum Lesen und eine zum Schreiben definiert. Daher lassen sich Variablen sowohl zur Erweiterung bereits implementierter Attribute verwenden als auch selbst wieder erweitern. Wir legen fest, daß die Version zum Lesen den durch die Grundversion gelieferten Wert ignoriert und den eigenen als Resultat an den Aufrufer weitergibt, während die Version zum Schreiben die Grundversion ausführt und dann den eigenen Wert ändert.

4.4 Kooperation unter Aspekten

Trotz der unterschiedlichen Aspekte, unter denen ein Objekt betrachtet werden kann, besitzt es eine Identität. Die verschiedenen Versionen der Komponenten arbeiten wenigstens teilweise auf einem gemeinsamen Zustand. Ein Benutzer, der eine `person` unter dem Aspekt des `Steuerzahler`s betrachtet, kann und soll daher beispielsweise feststellen, daß der Steuersatz entsprechend der Progression angeglichen wurde, wenn sich das Einkommen der `person` durch den Befehl eines anderen Benutzers gesteigert hat, der in der `person` einen `Arbeitnehmer` sieht. Dies ließe sich erreichen, indem die Sicht `Arbeitnehmer` eine Methodenversion des Schreibe-Befehls des Attributs `gehalt` verwendet, die zudem den Steuersatz angleicht. Wenn der Aspekt `Steuerzahler` allerdings erst zu einem späteren Zeitpunkt hinzukam, so bedeutet das, eine bereits vorhandene Methodenversion anzupassen und dazu erst einmal zu verstehen.

Wünschenswert ist hingegen, die zum Angleichen des Steuersatzes notwendigen Anwei-

```
1  OBJECT
     ...
     IMPLEMENTATION
       MAPPINGS
5        VIEW Person
           gehalt BY GrundVersion, SteuerzahlerVersion; ...
         END VIEW Person;
         VIEW Arbeitnehmer
           gehalt BY GrundVersion, SteuerzahlerVersion; ...
10       END VIEW Arbeitnehmer;
         VIEW Steuerzahler
           gehalt BY GrundVersion, SteuerzahlerVersion; ...
         END VIEW Steuerzahler

15     ATTRIBUTES
         ...
         ASSERTED gehalt: INTEGER BY
           VERSION GrundVersion BY VARIABLE END VERSION GrundVersion,
           VERSION SteuerzahlerVersion BY
20           PUT METHOD
               Steuersatz neu berechnen -> steuersatz
             END PUT
           END VERSION SteuerzahlerVersion
         END gehalt;
25       ASSERTED steuersatz: INTEGER BY VARIABLE

       COMMANDS
         ...
29   END OBJECT AS Person -> person
```

Beispiel 4.5.: Kooperation unter Aspekten

sungen getrennt angeben zu können. Allgemein gesprochen kann es notwendig sein, in der Abbildung einer Sicht nicht nur die Implementationsversion einer Komponente anzugeben, die dem durch die Sicht beschriebenen Aspekt entspricht, sondern auch Versionen anderer Aspekte, mit denen kooperiert werden soll. In ARISTARCH/L kann man zu diesem Zweck in der Abbildung einer Sicht mehrere Versionen für eine Komponente angeben, die in nicht festgelegter Reihenfolge mit denselben Argumenten ausgeführt werden. Auch bei der `EXTENDING`-Klausel ist die Angabe mehrerer Grundversionen möglich. Dieser Vorgang wird insgesamt als *Kombination von Versionen* bezeichnet. Es ist daher möglich, die unter dem Aspekt `Steuerzahler` notwendigen Anweisungen in einer Methodenversion getrennt von denen unter `Arbeitnehmer` zu beschreiben, aber durch Angabe beider Versionen zu erreichen, daß bei Steigerung des Einkommens auch der Steuersatz angeglichen wird.

Das Attribut `gehalt` besitzt daher zwei unterschiedliche Versionen, wie Beispiel 4.5 zeigt (Zeilen 17 bis 24). Die Grundversion beschreibt das Gehalt als Variable, die über die Sicht als Arbeitnehmer beispielsweise vom Arbeitgeber geändert werden kann. Zur Anpassung des Steuersatzes enthält die Steuerzahlerversion lediglich eine `PUT`-Methode für das Attribut `gehalt`. In den Abbildungen der Sichten werden beide Versionen als gleichberechtigt angeführt (Zeilen 6, 9 und 12). Damit ist gewährleistet, daß unabhängig von den Aspekten auch bei einer Änderung des Gehalts durch das Objekt selbst sowohl das Gehalt, als auch in Abhängigkeit davon der Steuersatz angepaßt wird. An diesem Beispiel wird außerdem deutlich, wie Attribute und Methoden zusammenspielen. Die Implementation als Variable definiert implizit eine `PUT`- und eine `GET`-Methode. Damit ist es für den Steuerzahleraspekt möglich, lediglich einen schreibenden Zugriff über eine eigene Methodenversion abzufangen und zur Anpassung des Steuersatzes zu verwenden (Zeilen 20 bis 22).

Werden mehrere erweiterte Versionen miteinander kombiniert, so werden die kombinierten Grundversionen durch die kombinierten Erweiterungen erweitert. Dabei muß allerdings gewährleistet sein, daß die Grundversionen nur genau einmal ausgeführt werden und zwar für jede

Erweiterung an der entsprechend markierten Stelle. Dazu werden in der entstehenden Methodenversion die Kommandosequenzen der Erweiterungen vor dem Schlüsselwort `BASE METHOD` vor den Grundversionen und die Sequenzen nach diesem Schlüsselwort nach den Grundversionen in nicht bestimmter Reihenfolge angeordnet.

Manchmal scheint die Nichtfestlegung der Reihenfolge der Kombination von Versionen kooperierender Aspekte ein Problem darzustellen. Nehmen wir beispielsweise an, ein Befehl soll zugleich Vergünstigungen und Modus der Gehaltsauszahlung eines Arbeitnehmer modifizieren. Jede dieser Modifikationen kann eine Angleichung des Steuersatzes notwendig machen. Dazu kann der Befehl durch eine entsprechende Version des Steuerzahleraspekts abgefangen werden. Allerdings muß diese Version ausgeführt werden, *nachdem* die entsprechenden Modifikationen stattgefunden haben, eine Forderung, die scheinbar im Widerspruch zu der Nichtfestlegung der Reihenfolge steht. Dieses Problem verschwindet jedoch, wenn man Versionen implementiert, die nicht den Befehl selbst abfangen, sondern die, die er zur Modifikation der Größen verwendet, die zu einer Angleichung des Steuersatzes führen können. Dies ist immer möglich, da die Methoden zum Lesen und Schreiben eines Attributs angepaßt werden können.

Die Nichtfestlegung der Reihenfolge der Kombination von Versionen ist ferner für die Berechnung von Seiteneffekten auf dem Objektzustand und den Resultaten von Bedeutung. Daher darf die wirkliche Reihenfolge die entstehenden Seiteneffekte nicht beeinflussen, die Modifikationen bezogen auf den Gesamtzustand des Objekts müssen kommutativ sein. Dies kann zwar durch die Sprache nicht erzwungen werden, ist jedoch in der Regel kein Problem, wie die folgenden Fälle zeigen.

- *Verschiedene Versionen modifizieren unterschiedliche Teile des Objektzustands.* Wenn die miteinander kombinierten Versionen auf unterschiedlichen Teilen des Objektzustands arbeiten, ist die Reihenfolge der Bearbeitung unerheblich. In Beispiel 4.5 bezieht sich so die `SteuerzahlerVersion` des `gehalts` lediglich auf das Attribut `steuersatz`, das erst mit dem Aspekt des Steuerzahlers eingeführt wird. Ähnliches gilt für die entsprechende Version der Methode für den Befehl `Status`. Die Funktionalität der anderen Versionen auf der gleichen Ebene ist für die betrachtete Version belanglos, da jede Version nur die ihren Aspekt betreffenden Elemente verwendet. Die Grundversion behandelt die allen Aspekten gemeinsamen Zustandsteile, die Stelle ihrer Abarbeitung ist jedoch durch Angabe des Schlüsselworts `BASE METHOD` festgelegt, so daß eine definierte Einbettung ihrer Seiteneffekte gewährleistet ist. Somit muß beim Hinzufügen weiterer Versionen zwar die Spezifikation der Gesamtoperation, nicht aber die Implementation der bisherigen Versionen berücksichtigt werden.

  Im Fall der Berechnung der Resultate, die letztlich Seiteneffekte auf den Ausgabeparametern sind, die als allen Versionen gemeinsame Attribute einer Ausführung betrachtet werden, erlaubt dies die Kombination von Versionen, die bis auf eine keinen Beitrag zu den Resultaten leisten. Diese Versionen entsprechen im allgemeinen Aspekten, die mit dem betrachteten Aspekt kooperieren, dienen also nur dem Abfangen eines Befehls zur Manipulation eines eigenständigen Teilzustands. In Beispiel 4.5 bezieht sich so die `SteuerzahlerVersion` des `gehalts` lediglich als Seiteneffekt auf das dem Aspekt des Steuerzahlers zugeordnete Attribut `steuersatz`, ist jedoch nicht am Auslesen des Attributwertes beteiligt.

- *Verschiedene Versionen modifizieren gemeinsame Teile des Objektzustands.* Die Reihenfolge der Seiteneffekte ist belanglos, wenn die Modifikationen der gemeinsamen Teile des Objektzustands kommutativ sind. Dies gilt in besonderer Weise im Fall der Berechnung der Resultate als allen Versionen gemeinsame Attribute einer Ausführung. In Beispiel 4.4

ergibt sich das Resultat durch die Und-Verknüpfung der Konsistenzprüfungen der einzelnen Versionen. Durch Vorbelegung des Resultatattributs durch das neutrale Element der Und-Verknüpfung ergibt sich insgesamt eine kommutative Modifikation, so daß die Reihenfolge der Abarbeitung der einzelnen Versionen belanglos ist.

Nicht kommutative Seiteneffekte auf demselben Teil des Objektzustands sollten dann durch gemeinsame Grundversionen für den betrachteten Befehl erzeugt werden. Dabei ist in ARISTARCH/L gewährleistet, daß Grundversionen genau einmal ausgeführt werden, auch wenn sie in verschiedenen Erweiterungspfaden auftreten. Bei der gleichzeitigen Entwicklung verschiedener Versionen muß explizit auf die Ausfaktorisierung von Zustandsmodifikationen geachtet werden, die mehrere Versionen gemeinsam besitzen. Die Ausfaktorisierung von Seiteneffekten ergibt sich bei nachträglichen Erweiterungen in der Regel jedoch von selbst, da sich jede Erweiterungsversion auf andere, eigenständige, in der bisherigen Ausprägung noch nicht vorhandene oder berücksichtigte Teile des Objektzustandes bezieht. Die Stelle der Ausführung der Grundversion wird durch Angabe des Schlüsselworts `BASE METHOD` festgelegt; für die Erweiterungsversionen, deren Reihenfolge nicht festgelegt ist, gilt dann der erste Fall.

4.5 Ableiten von Sichten

Oft beinhaltet ein Aspekt eines Objekts einen anderen, stellt also eine Spezialisierung dar. Beispielsweise möchte man einen `Arbeitnehmer` insbesondere als `Person` betrachten. Entsprechend erlaubt ARISTARCH/L das Ableiten von Sichten auseinander. Die in den Schnittstellen der Grundsichten angegebenen Komponenten werden dabei im allgemeinen in die Schnittstelle der abgeleiteten Sicht übernommen. Ausgenommen davon sind lediglich die Komponenten mit den in der `OMITTING`-Klausel angegebenen Namen. Die durch die abgeleitete Sicht angegebenen Komponenten erweitern die so entstandene Grundschnittstelle.

Anspruchsvoller als das Vereinigen der Schnittstellen ist hingegen die Behandlung der Abbildungen. Im Vordergrund steht hier wieder die Idee von Namen als Spezifikationen. Die durch die Abbildungen der Grundsichten gegebenen Implementationsversionen einer Komponente, die *Grundversionen*, werden daher miteinander kombiniert. Dies leuchtet ein, denn die durch die Grundsichten dargestellten Aspekte kooperieren in dem durch die abgeleitete Sicht dargestellten Aspekt miteinander. Werden in der Abbildung der abgeleiteten Sicht weitere Versionen angegeben, so werden sie als aspektspezifische Erweiterungen betrachtet.

In Beispiel 4.6 wird ausgenutzt, daß die Sichten `Arbeitnehmer` und `Steuerzahler` jeweils die Sicht `Person` umfassen sollen. Daher werden beide Sichten von der Sicht `Person` abgeleitet und übernehmen damit die allgemeine Personenschnittstelle (Zeilen 11 und 17). Gleichzeitig wird auch die Abbildung dieser Sicht übernommen, so daß bei der Definition der Erweiterungssichten die jeweils angegebene Version für den `Status`- und den `Konsistenz`-Befehl implizit die Grundversion der Grundsicht `Person` erweitert. Die Angabe der `EXTENDING`-Klausel ist damit überflüssig. Der Aufbau der verschiedenen Versionen entspricht dem des vorangegangenen Abschnitts.

Wie bei der direkten Erweiterung von Versionen kann demnach die gleiche Grundversion genutzt werden, um verschiedene Implementationen mit unterschiedlichen Erweiterungen zu realisieren, ohne den entsprechenden Kode vervielfältigen zu müssen. Umgekehrt ist es nun aber auch möglich, die gleiche Erweiterung über verschiedene Sichten mit unterschiedlichen Grundversionen zu verknüpfen. Dazu muß die Erweiterung nur in mehreren Sichten angegeben werden, die von Sichten mit unterschiedlichen Grundversionen ableiten.

Gleichzeitig genügt es, die Abbildung des Attributs `gehalt` lediglich in der Grundsicht zu definieren (Zeile 29), da sie in alle abgeleiteten Sichten übernommen wird. Einfach ist jetzt auch

```
1  OBJECT
     INTERFACES
       VIEW Person
         ATTRIBUTES
5          IMMUTABLE name: tString
         COMMANDS
           Status ();
           Aktion ()
       END VIEW Person;
10
       VIEW Arbeitnehmer DERIVED FROM Person
         ATTRIBUTES
           personalnummer: INTEGER;
           gehalt: INTEGER
15     END VIEW Arbeitnehmer;

       VIEW Steuerzahler DERIVED FROM Person
         ATTRIBUTES
           steuernummer: INTEGER;
20         IMMUTABLE steuersatz: INTEGER
       END VIEW Steuerzahler;

       VIEW SteuerzahlenderArbeitnehmer DERIVED FROM Arbeitnehmer, Steuerzahler
       END VIEW SteuerzahlenderArbeitnehmer
25
     IMPLEMENTATION
       MAPPINGS
         VIEW Person
           gehalt BY GrundVersion, SteuerzahlerVersion;
30         Status BY GrundVersion;
           Konsistenz BY GrundVersion
         END VIEW Person;
         VIEW Arbeitnehmer
           Status BY ArbeitnehmerVersion;
35         Konsistenz BY ArbeitnehmerVersion
         END VIEW Arbeitnehmer;
         VIEW Steuerzahler
           Status BY SteuerzahlerVersion;
           Konsistenz BY SteuerzahlerVersion
40       END VIEW Steuerzahler

       ATTRIBUTES
         ...
       COMMANDS
45       ...
   END OBJECT AS Person -> person
```

Beispiel 4.6.: Ableiten von Sichten

das Zusammenführen aller Sichten zu einer globalen Sicht, die einen vollständigen Überblick über das Objekt `person` erlaubt, indem als Ableitung von den beiden Sichten `Arbeitnehmer` und `Steuerzahler` eine weitere Sicht `SteuerzahlenderArbeitnehmer` vereinbart wird (Zeilen 23 bis 24). Die korrekte Implementation ergibt sich durch Kombination und Erweiterung der einzelnen Implementationsversionen. Insbesondere gilt dies für die Implementation der Befehle `Status` und `Konsistenz`, die unter diesem Aspekt alle Teilversionen einschließlich der Grundversion enthält. Man beachte, daß der Befehl `Konsistenz` aus Beispiel 4.4 auch in Beispiel 4.6 problemlos arbeitet, da die Operation `AND` kommutativ ist.

4.6 Sichten und Delegation

Delegation dient der Behandlung sehr unterschiedlicher Probleme, und in dieser Polyvalenz liegt unsere Kritik. Diese Probleme spiegeln sich in bereits oben beschriebenen Forderungen wider und sollen im Zusammenhang mit dem Modell von ARISTARCH/L genauer betrachtet werden.

Eines dieser Probleme ist die Beschreibung von Objekten, die von verschiedenen Anwen-

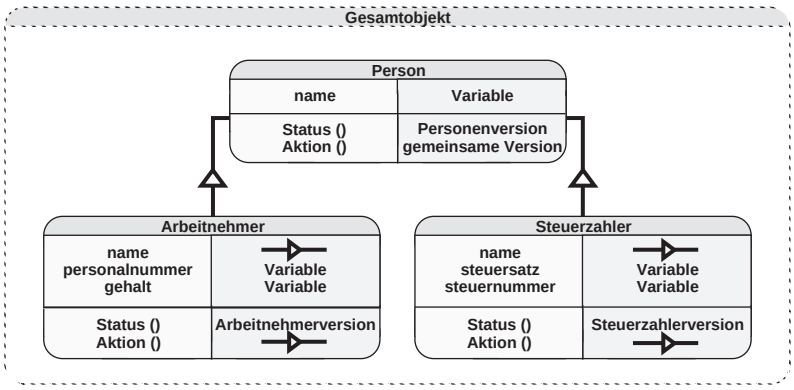

Bild 4.2: Sichten mittels Delegation

dungen unter unterschiedlichen Aspekten betrachtet werden. Delegation kann, wie in Bild 4.2 gezeigt, dazu benutzt werden, um diese Aspekte durch verschiedene Objekte darzustellen, die durch ein oder mehrere gemeinsame Prototypen über einen wenigstens teilweise gemeinsamen Zustand verfügen. Das so entstehende Objektgeflecht stellt eigentlich ein einziges Objekt dar. Dony, Malenfant und Cointe [6] bezeichnen ein solches Objekt folgerichtig als *Split Object*. Die die unterschiedlichen Aspekte darstellenden Objekte können mit Sichten in ARISTARCH/L verglichen werden. Delegation übernimmt dabei die Aufgabe der Abbildungen. Dann sind beispielsweise `Steuerzahler` und `Arbeitnehmer` zwei Objekte, die an ein Objekt `Person` delegieren, und gemeinsam mit diesem das *Split Object* `person` darstellen. Alle drei Objekte besitzen unterschiedliche Methoden für den Befehl `Status`. Durch das *extended self* (Lieberman [12]) wird unabhängig vom Ort des Aufrufs von `Status` in einem dieser drei Objekte immer die Methode des Objekts aufgerufen, das den Aspekt darstellt, unter dem der Aufruf stattfindet. So wird die Methode `Status` von `Steuerzahler` auch dann verwendet, wenn sie aus einer Methode des Grundobjekts `Person` für einen von `Steuerzahler` delegierten Befehl aufgerufen wurde. Besitzen mehrere Sichten eines Objekts dieselbe Implementationsversion für eine Komponente, so kann das durch das Einsetzen eines weiteren Objekts zwischen `Person` und den den Sichten entsprechenden Objekten dargestellt werden, das diese Version realisiert. Ein schwerwiegender Nachteil der Delegation beim Beschreiben verschiedener Aspekte eines Objekts ist, daß die Betrachtung des *Split Object* als logisches Ganzes nicht unterstützt wird. Andererseits widerspricht die Verwendung der Delegation zur Beschreibung von *Split Objects* auch der Idee des prototypbasierten Ansatzes, indem nicht einzelne Objekte, sondern verschiedene Aspekte eines einzigen Objekts beschrieben werden. In ARISTARCH/L hingegen bezeichnet eine Referenz stets das gesamte Objekt, gibt aber zusätzlich an, unter welchem Aspekt dieses Objekt gerade betrachtet wird.

Ein weiterer Nachteil ist, daß Delegation zugleich die dynamische Erweiterung eines *Split Objects* um weitere Aspekte erlaubt. Die dynamische Änderung von Objekten allgemein, also die Modifikation bestehender Objekte, aber ist orthogonal zur Unterstützung unterschiedlicher Aspekte und sollte daher durch ein zu Sichten orthogonales Sprachelement realisiert werden. Delegation dient somit im Fall der *Split Objects* letztlich der Erfüllung zweier getrennter Forderungen. Dies ist ein Indiz für die Richtigkeit unserer Ansicht, daß sie einen Mechanismus für die Implementation bestimmter Elemente einer Sprache darstellen kann, aber selbst nicht auf der Sprachebene angeboten werden sollte. Die Unterstützung von Objekten mit unterschiedlichen Aspekten einerseits und die dynamische Änderung von Objekten andererseits stellen getrennte Probleme dar, und sie sollten daher mit verschiedenen Sprachelementen realisiert werden. Szyperski [29] bringt dies auf die Kernaussage, daß

„It is a good rule of thumb to keep the number of concepts in a language small. However, it is always possible to reduce everything to (almost) a single concept. For example, in some functional languages everything is expressed using the single concept of higher-order functions. This tends to make programs hard to grasp. Keeping orthogonality and completeness in mind, it is often preferable to provide *separate concepts for separate problems*. This leads to more natural ways of expressing solutions in a given language."

Das *Single Concept* in delegationsbasierten Sprachen ist das delegierende Objekt.

5 Kategorisierung von Objekten

Die bisher beschriebenen Sprachelemente erlauben die Beschreibung von einzelnen Objekten aus sich selbst heraus. Nach den in Kapitel 2 aufgestellten Forderungen aber muß eine objektbasierte Sprache sowohl die konkrete als auch die abstrakte Kategorisierung von Objekten erlauben.

Die konkrete Kategorisierung eines Objekts wird in ARISTARCH/L durch *Klonieren* genutzt. Dabei erbt das durch Klonieren erzeugte Objekt die Sichten, die Komponenten und den Zustand eines oder mehrerer Prototypen. Dem weiteren liegt wieder die Idee von Namen als Spezifikationen zugrunde. Eine Sicht eines bestimmten Namens des erzeugten Objekts ist immer implizit abgeleitet von allen Sichten gleichen Namens der Prototypen. Nach dem Klonieren sind das neue Objekt und seine Vorlagen unabhängig voneinander modifizierbar. Dies gilt nicht nur für den Zustand, sondern insbesondere auch für die Sichten und Komponenten.

In klassenbasierten Sprachen wird die abstrakte Kategorisierung von Objekten dadurch erreicht, daß alle Objekte Exemplare einer Klasse sind. In ARISTARCH/L können Objekte hierzu beliebig viele *Exemplarmuster* oder kurz *Muster* besitzen, die zu Objekten exemplifiziert werden können.

5.1 Muster

Muster beschreiben Mengen gleichartiger Objekte. Dazu besitzen sie dieselben Elemente wie Beschreibungen konkreter Objekte, also die Komponenten und ihre Implementationsversionen und verschiedene Sichten. Aus einem Objekt mit verschiedenen Mustern können verschiedene Objekte entsprechend dieser Muster erzeugt werden. Objekte mit Mustern stellen daher Klassen dar, die aber verschiedene Arten von Exemplaren erzeugen können. Obwohl Klassen Objekte sind, ergibt sich aber kein unendlicher Metaregreß, weil Objekte direkt beschrieben werden können.

Aus dem bereits beschriebenen Objekt `person` wird in Beispiel 5.1 ein Muster eines Objekts. Der `INTERFACES`-Abschnitt des Objekts `person` wird in den `PATTERNS`-Abschnitt der Sicht des `INTERFACES`-Abschnitts des neuen Objekts übernommen (Zeilen 5 bis 7). Entsprechend wird für den `IMPLEMENTATION`-Abschnitt vorgegangen (Zeilen 12 bis 14). Über einen `NEW`-Ausdruck können dann, wie in Beispiel 5.2 gezeigt, beliebig viele diesem Muster entsprechende Exemplare erzeugt werden. Beschreibt ein Muster, wie in diesem Beispiel, mehrere Sichten, muß neben dem Objekt, dessen Muster exemplifiziert werden soll, mittels einer `AS`-Klausel auch die initiale Sicht bestimmt werden, unter der das Objekt betrachtet werden soll.

Ein Muster kann ebenso wie Methoden verschiedene Versionen besitzen, die zur Implementation des Musters in verschiedenen Sichten verwendet werden. Damit können zum Beispiel über verschiedene Sichten entsprechend unterschiedlicher Randbedingungen optimierte Versionen einer Datenstruktur bereitgestellt werden. So lassen sich etwa zwei Versionen einer Liste formulieren, die den schnellen Zugriff auf beliebige Elemente oder aber das schnelle

```
1 OBJECT
    INTERFACES
      VIEW
        PATTERNS
5         PersonenMuster INTERFACES
            ...
        END PersonenMuster
      END VIEW

10  IMPLEMENTATION
      PATTERNS
        ASSERTED PersonenMuster IMPLEMENTATION
          ...
        END PersonenMuster
15 END OBJECT -> personenklasse
```

Beispiel 5.1.: Beschreibung von Exemplarmustern

```
NEW PersonenMuster OF personenklasse AS Arbeitnehmer -> arbeitnehmer
```

Beispiel 5.2.: Erzeugung von Exemplaren

Einfügen weiterer Elemente erlauben. Dies ist in Beispiel 5.3 zu sehen. Die abstrakte Sicht `Liste` (Zeilen 3 bis 13) dient zur einheitlichen Beschreibung der beiden abgeleiteten Sichten `ListeMitSchnellemZugriff` und `ListeMitSchnellemEinfuegen` (Zeilen 14 und 15). Abstrakte Sichten müssen nicht für jede Komponente eine Abbildung besitzen und können daher auch nicht zur Benutzung eines Objekts verwendet werden. Sie erlauben jedoch, wie in diesem Beispiel gezeigt, die Beschreibung gemeinsamer Definitionen von Sichten mit unterschiedlichen Abbildungen (Zeilen 20 und 23), ohne für diese Sichten eine gemeinsame Grundversion vorzugeben. Obwohl beide Sichten daher insbesondere dasselbe Muster `ListenMuster` anbieten, wird es einmal durch die Version `SchnellerZugriffVersion` implementiert, die schnellen Zugriff auf bestimmte Elemente bietet, indem die Liste durch eine Reihung der Elemente dargestellt wird (Zeilen 28 bis 34), während die Version `SchnellesEinfuegenVersion` durch die Verkettung der Elemente sowohl schnelles Einfügen weiterer Elemente am Anfang der Liste als auch schnelles Anfügen am Ende auf Kosten des Direktzugriffs erlaubt (Zeilen 35 bis 42).

Auch die Erweiterung einer Implementationsversion eines Musters durch eine andere ist möglich. Sie wird auf die Erweiterung der Komponenten entsprechend der Abbildungen der Sichten in den beiden Versionen zurückgeführt.

5.2 Ableiten von Mustern

Um Gemeinsamkeiten zwischen Mustern zu beschreiben, lassen sie sich ebenso wie Sichten auseinander ableiten. Sichten der Grundmuster werden im allgemeinen übernommen. Ausgenommen hiervon sind lediglich die Sichten mit den in der `OMITTING`-Klausel des abgeleiteten Musters angegebenen Namen. Dem weiteren liegt wieder die Idee von Namen als Spezifikationen zugrunde. Eine Sicht eines bestimmten Namens des abgeleiteten Musters ist immer implizit abgeleitet von allen Sichten gleichen Namens der Grundmuster. Das Ableiten von Mustern wird somit auf das Ableiten von Sichten reduziert. Alle anderen Sichten des abgeleiteten Musters müssen explizit direkt oder indirekt von wenigstens einer Sicht jedes Grundmusters abgeleitet werden, denn andernfalls kann nicht immer garantiert werden, daß jeder Komponente eine Implementation zugeordnet ist.

Die in Abschnitt 4 beschriebenen Regeln zum impliziten Zuordnen von Implementationen

```
OBJECT
  INTERFACES
    ABSTRACT VIEW Liste
      PATTERNS
        ListenMuster INTERFACES
            VIEW
              COMMANDS
                Zugreifen (INTEGER) -> (tElem);
                Einfuegen (tElem);
                Anhaengen (tElem)
            END VIEW
        END ListenMuster
    END VIEW Liste;
    VIEW ListeMitSchnellemZugriff DERIVED FROM Liste END VIEW ListeMitSchnellemZugriff;
    VIEW ListeMitSchnellemEinfuegen DERIVED FROM Liste END VIEW ListeMitSchnellemEinfuegen

  IMPLEMENTATION
    MAPPINGS
      VIEW ListeMitSchnellemZugriff
        ListenMuster BY SchnellerZugriffVersion
      END VIEW ListeMitSchnellemZugriff;
      VIEW ListeMitSchnellemEinfuegen
        ListenMuster BY SchnellesEinfuegenVersion
      END VIEW ListeMitSchnellemEinfuegen

    PATTERNS
      ASSERTED ListenMuster IMPLEMENTATION
        VERSION SchnellerZugriffVersion
          ATTRIBUTES
            Repräsentation durch Reihung
          COMMANDS
            ASSERTED Zugreifen (n: INTEGER) BY METHOD ... n. Element liefern;
            ...
        END VERSION SchnellerZugriffVersion,
        VERSION SchnellesEinfuegenVersion
          ATTRIBUTES
            Repräsentation durch Verkettung
          COMMANDS
            ...
            ASSERTED Einfuegen (elem: tElem) BY METHOD ... Neues erstes Element einketten;
            ASSERTED Anhaengen (elem: tElem) BY METHOD ... Neues letztes Element einketten
        END VERSION SchnellesEinfuegenVersion
      END ListenMuster
END OBJECT
```

Beispiel 5.3.: Musterversionen

gemeinsam mit der Regel des vorangegangenen Absatzes, die Sichten der beteiligten Muster mit gleichem Namen zu vereinigen, erlauben es, eine typische Konstellation mit geringem Aufwand zu beschreiben, in der jedes Exemplarmuster nur eine Sicht eines festgelegten Namens beschreibt, und jede Komponente nur eine Implementationsversion besitzt. Die Regeln bewirken dann, daß in keiner Sicht eine explizite Abbildung notwendig ist, die Implementationsversion im abgeleiteten Muster immer verwendet wird, und ferner die Sicht des abgeleiteten Musters sich durch Vereinigung mit denen der Grundmuster ergibt. Die genannte Konstellation liegt immer vor im Fall objektorientierter Sprachen herkömmlicher Art wie beispielsweise C++ (Ellis und Stroustrup [7]). Der Mehraufwand in ARISTARCH/L ist bei typischen Konstellationen demnach gering.

Die Sichtenhierarchie aus Beispiel 4.6 kann jetzt wie in Beispiel 5.4 als Ausgangspunkt für eine Musterhierarchie dienen. Die verschiedenen Implementationsversionen verteilen sich dann auf die entsprechenden Muster, so daß auf die explizite Definition von verschiedenen Implementationsversionen innerhalb eines Musters verzichtet werden kann. Die so entstandene Klasse enthält jetzt mehrere Muster, die separat exemplifiziert werden können, sofern sie, wie in diesem Beispiel, alle an mindestens einer Schnittstelle einer Sicht angeboten werden.

```
1 OBJECT
    INTERFACES
      VIEW
        PATTERNS
5         PersonenMuster INTERFACES
            VIEW Person ...
          END PersonenMuster;

          ArbeitnehmerMuster DERIVED FROM PersonenMuster INTERFACES
10          VIEW Arbeitnehmer DERIVED FROM Person ...
          END ArbeitnehmerMuster;

          SteuerzahlerMuster DERIVED FROM PersonenMuster INTERFACES
            VIEW Steuerzahler DERIVED FROM Person ...
15        END SteuerzahlerMuster;

          SteuerzahlenderArbeitnehmerMuster DERIVED FROM ArbeitnehmerMuster, SteuerzahlerMuster
          END SteuerzahlenderArbeitnehmerMuster
      END VIEW
20
    IMPLEMENTATION
        PATTERNS
          ASSERTED PersonenMuster IMPLEMENTATION ...
            COMMANDS
25            ASSERTED Status () BY METHOD ... Grundversion;
              ASSERTED Aktion () BY METHOD ...;
              Konsistenz () -> (TRUE -> konsistent: BOOLEAN) BY METHOD ...Grundversion
          END PersonenMuster;
          ASSERTED ArbeitnehmerMuster IMPLEMENTATION ...
30          COMMANDS
              ASSERTED Status () BY METHOD ... Arbeitnehmeranteil;
              Konsistenz () -> (TRUE -> konsistent: BOOLEAN) BY METHOD ...Arbeitnehmeranteil
          END ArbeitnehmerMuster;
          ASSERTED SteuerzahlerMuster IMPLEMENTATION ...
35          COMMANDS
              ASSERTED Status () BY METHOD ... Steuerzahleranteil;
              Konsistenz () -> (TRUE -> konsistent: BOOLEAN) BY METHOD ...Steuerzahleranteil
          END SteuerzahlerMuster
39  END OBJECT -> personenklasse
```

Beispiel 5.4.: Ableitung von Mustern

Das `SteuerzahlenderArbeitnehmerMuster` enthält die vollständige, aus den vorangegangenen Beispielen bekannte Beschreibung einer Person (Zeilen 17 bis 18), während sich die anderen Muster auf Teilmengen beschränken.

Das Beispiel zeigt auch die nicht zugesicherte Komponente `Konsistenz` im `PersonenMuster` (Zeile 27). Durch ein Muster ist nur die Existenz von `ASSERTED`-Komponenten garantiert, `Konsistenz` kann also nicht in abgeleiteten Mustern namentlich verwendet werden. Im Beispiel trifft dies für das `SteuerzahlenderArbeitnehmerMuster` zu. Implementiert dagegen ein abgeleitetes Muster, wie beispielsweise `ArbeitnehmerMuster`, selbst eine solche Komponente, stellt sie entsprechend der Idee von Namen als Spezifikationen eine Erweiterung der Grundversion dar (Zeile 32). Da bei einer Versionserweiterung jedoch nicht explizit auf den Namen Bezug genommen wird, bleibt die Grundversion dem abgeleiteten Muster verborgen, und die Implementation des abgeleiteten Musters bleibt korrekt, unabhängig davon, ob das Grundmuster einen Beitrag zur Implementation der Komponente leistet oder nicht. Im vorliegenden Fall enthält damit die Konsistenzprüfung des `ArbeitnehmerMuster`s implizit die Konsistenzprüfung des Grundmusters `PersonenMuster`. Man beachte, daß durch abgeleitete Muster die Implementation einer Komponente auf mehrere `IMPLEMENTATION`-Abschnitte verteilt sein kann. Jedes Muster kann demnach eigene Initialwerte für Befehlsresultate bei seinen Implementationsteilen bestimmen. Beim Aufruf eines Befehls werden die Werte des Musters verwendet, aus dem das beauftragte Objekt entstanden ist.

Ein Muster kann von einem anderen Muster abgeleitet werden, dem in verschiedenen Sich-

```
1  OBJECT
     INTERFACES
       ABSTRACT VIEW Liste
         PATTERNS
5          ListenMuster ... END ListenMuster;
           ErweitertesListenMuster DERIVED FROM ListenMuster
             COMMANDS
               LetztesElement () -> (tElem)
           END ErweitertesListenMuster
10     END VIEW Liste;
       VIEW ListeMitSchnellemZugriff DERIVED FROM Liste END VIEW ListeMitSchnellemZugriff;
       VIEW ListeMitSchnellemEinfuegen DERIVED FROM Liste END VIEW ListeMitSchnellemEinfuegen

     IMPLEMENTATION
15     MAPPINGS
         VIEW ListeMitSchnellemZugriff
           ListenMuster BY SchnellerZugriffVersion
         END VIEW ListeMitSchnellemZugriff;
         VIEW ListeMitSchnellemEinfuegen
20         ListenMuster BY SchnellesEinfuegenVersion
         END VIEW ListeMitSchnellemEinfuegen

       PATTERNS
         ...
25       ASSERTED ErweitertesListenMuster IMPLEMENTATION
           COMMANDS
             ASSERTED LetztesElement () -> (tElem) BY METHOD ...
         END ErweitertesListenMuster
29   END OBJECT
```

Beispiel 5.5.: Musterversionen und Oder-Vererbung

ten unterschiedliche Implementationsversionen zugeordnet sind. Wird es in mehreren Sichten angeboten, die von Sichten mit unterschiedlichen Implementationsversionen für das Grundmuster ableiten, ist das Muster in diesen Sichten von unterschiedlichen Grundmusterversionen abgeleitet. Dies entspricht gerade der von LaLonde, Pugh und Thomas [11] beschriebenen Oder-Vererbung. Beispielsweise möchten wir die in Beispiel 5.3 beschriebenen Versionen um einen dedizierten Befehl zum Zugriff auf das letzte Element der Liste erweitern. Das Ergebnis zeigt Beispiel 5.5. Der erweiterte Funktionsumfang wird durch das Muster `ErweitertesListenMuster` beschrieben, das vom Grundmuster `ListenMuster` abgeleitet ist (Zeilen 6 bis 9). Es wird wie in Beispiel 5.3 durch Ableitung von der abstrakten Sicht `Liste` zusammen mit dem Grundmuster `ListenMuster` in den beiden abgeleiteten Sichten `ListeMitSchnellemZugriff` und `ListeMitSchnellemEinfuegen` angeboten (Zeilen 11 und 12). Durch die unterschiedlichen Implementationsversionen für das Grundmuster in diesen beiden Sichten ist das nur in einer Implementationsversion beschriebene Muster `ErweitertesListenMuster` im einen Fall von Version `SchnellerZugriffVersion` und im anderen Fall von Version `SchnellesEinfuegenVersion` von `ListenMuster` abgeleitet. Die Abbildung der Implementationsversion des neuen Musters `ErweitertesListenMuster` geschieht implizit.

5.3 Exemplare und Klassenkomponenten

Klassenkomponenten, also solche eines Objekts mit Mustern, werden zugleich als allen Exemplaren gemeinsame Komponenten betrachtet. Diese Wirkung ist im wesentlichen bestimmt, indem man jedes Muster als von der Klasse abgeleitet betrachtet. Unterschiede zur Ableitung von einem Muster mit denselben Komponenten bestehen allerdings in zweierlei Hinsicht. Zum einen sind die Komponenten der Klasse nur einmal vorhanden. Zum anderen werden weder Sichten der Klasse in das Muster noch die Schnittstellen von Klassensichten in abgeleitete Exemplarsichten übernommen. Dies entspricht der Angabe aller Klassensichten in der `OMITTING`-Klausel

```
1 OBJECT
    INTERFACES
      VIEW
        PATTERNS
5         ZeichenkettenklassenMuster INTERFACES
            VIEW
              PATTERNS
                ZeichenkettenMuster ...
            END VIEW;
10        END ZeichenkettenklassenMuster
      END VIEW

    IMPLEMENTATION
      PATTERNS
15      ASSERTED ZeichenkettenklassenMuster IMPLEMENTATION
          ATTRIBUTES
            ASSERTED Zeichenfolgentabelle: tTabelle
          PATTERNS
            ASSERTED ZeichenkettenMuster ...
20      END ZeichenkettenklassenMuster
  END OBJECT
```

Beispiel 5.6.: Unabhängige Zeichenkettenklassen mittels geschachtelter Muster

des Musters und der Angabe aller in den Schnittstellen angegebenen Komponenten in der `OMITTING`-Klausel der Exemplarsicht. Die Implementationsversion einer Klassenkomponente, die durch ein Exemplar benutzt wird, ist demnach bestimmt durch die aktive Exemplarsicht. Jede Sicht eines Musters muß daher direkt oder indirekt von einer Sicht der Klasse abgeleitet sein.

5.4 Geschachtelte Muster

ARISTARCH/L unterstützt Klassen von Exemplaren, die ihrerseits als Klassen dienen, durch beliebig tief *geschachtelte* Muster. Geschachtelte Muster können innerhalb eines Musters ebenfalls voneinander abgeleitet werden. Sie werden aber auch wie alle anderen Komponenten eines Musters an abgeleitete Muster vererbt. Damit ist auch die Ableitung von Mustern möglich, die in ein anderes als das eigene Muster geschachtelt sind. Letztlich ist damit die Klassifikation von Musterhierarchien möglich.

Mittels geschachtelter Muster läßt sich nun eine Lösung des anfänglich geschilderten Problems der Zeichenkettenklassen mit unterschiedlicher Zustandsinformation angeben. Die Zeichenkettenklasse enthält als gemeinsamen Zustand aller ihrer Exemplare eine Tabelle für die Zeichenfolgen. In Beispiel 5.6 wird diese Klasse nochmals klassifiziert (Zeilen 5 bis 10 und 15 bis 20), so daß sich ein Objekt ergibt, das ein Muster enthält, das zu Objekten exemplifiziert werden kann, die schließlich ein Muster für die Zeichenketten-Objekte enthalten (Zeilen 8 und 19). Da die `Zeichenfolgentabelle` in dem Muster der ersten Ebene vereinbart ist, das die Zeichenkettenklasse beschreibt, entsteht mit jeder Exemplifikation ein Objekt mit eigenständigem Tabellenzustand, an dem mittels eines `NEW`-Ausdrucks Zeichenketten-Objekte erzeugt werden können.

Ein anderes Beispiel, dargestellt in Bild 5.1, soll im folgenden lediglich skizziert werden, da es für eine ausführliche Beschreibung zu umfangreich ist. Zugriffsobjekte für Dateien kann man als Exemplare der entsprechenden Datei betrachten, die implizit Zugriff auf die in der Datei enthaltene Information haben. Sie erlauben den zustandsbehafteten Zugriff auf Dateien, indem sie beispielsweise den für einen bestimmten Benutzer aktuellen Datensatzindex festhalten und stellen demnach die *Rollen* dar, die die Datei dem Benutzer gegenüber spielt. Zugriffsobjekte besitzen als Rollen eine eigene Identität, stehen aber in enger Beziehung zu der zugrundeliegenden

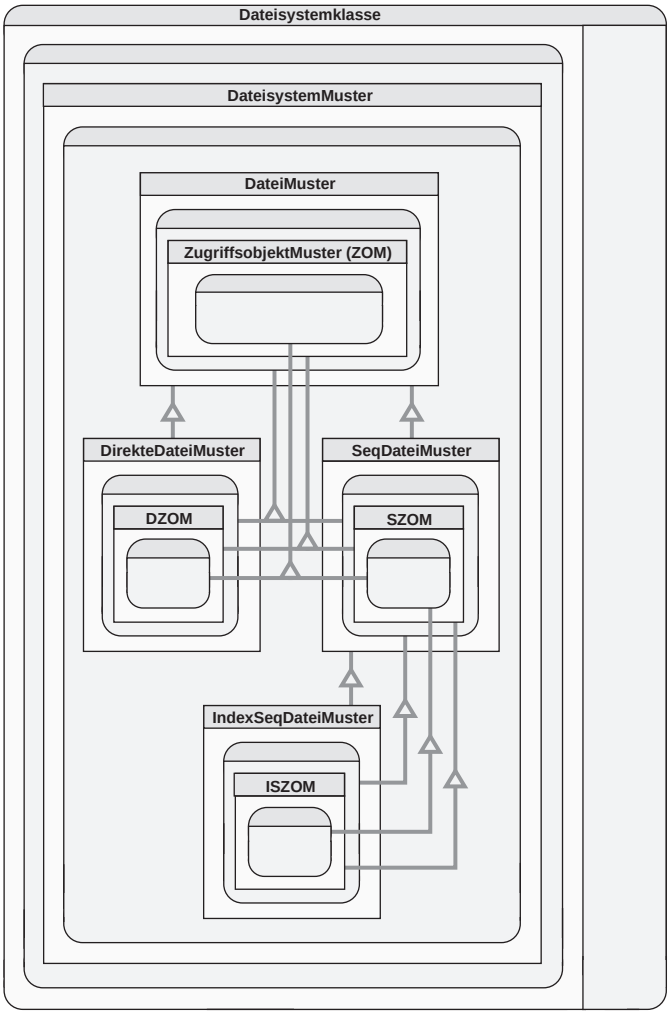

Bild 5.1: Beschreibung eines Dateisystems mittels geschachtelter Muster

Datei. Rollen im Sinne der Forderungen in Abschnitt 2.3 werden demnach durch Exemplarmuster beschrieben und sind damit in beliebiger Anzahl bereitstellbar. Letztlich bedeutet dann das Öffnen einer Datei das Erzeugen eines Zugriffsobjekts oder einer Rolle der Datei entsprechend dem enthaltenen Muster. Es kann nun unterschiedliche Arten von Dateien in einem Dateisystem geben, etwa sequentielle, indexsequentielle und direkte Dateien. Indexsequentielle Dateien sind Spezialisierungen von sequentiellen Dateien und werden daher durch Ableitung ihres Musters von dem sequentieller Dateien beschrieben. Alle Dateimuster wiederum sind von einem generischen Dateimuster abgeleitet, das unter anderem ein Muster der allgemeinen Zugriffsobjekte enthält, das daher alle Dateimuster erben. Alle Dateien eines Dateisystems insgesamt werden als Exemplare dieses Dateisystems aufgefaßt, die implizit Zugriff auf die Verwaltungsinformation des Dateisystems haben, also auf den Zustand des das Dateisystem darstellenden Objekts. Bis jetzt haben wir demnach mit dem Dateisystem ein Objekt, das eine Anzahl von Mustern besitzt, die selbst wiederum ein Muster enthalten. Da es im allgemeinen mehrere Datenträger mit Dateisystemen gibt, die jeweils einen eigenen Zustand haben, ist es sinnvoll, mittels eines Objekts mit einem entsprechenden Dateisystemmuster eine Klasse der Dateisysteme zu definieren. Auf diese Weise ergibt sich schließlich ein Objekt mit Mustern erster, zweiter und dritter Ordnung. Wie im Bild gezeigt, sind sie jeweils über Sichten des Musters der nächst höheren

```
1 OBJECT IS PersonenMuster OF personenklasse
    INTERFACES
      VIEW
        COMMANDS
5         SchneideGrimassen ()
      END VIEW

    IMPLEMENTATION
      ...
10END OBJECT -> JerryLewis
```

Beispiel 5.7.: Erweiterung von Exemplaren

Ordnung ineinander geschachtelt.

Das vorangegangene Beispiel zeigt deutlich, in welcher Weise geschachtelte Muster die Funktionalität bisheriger objektorientierter Sprachen erweitern. Bei jedem Aufruf eines in einer solchen Sprache geschriebenen Programms werden die enthaltenen Klassen mit einem neuen Zustand erzeugt. Programme können damit als Klassen zweiter Ordnung betrachtet werden. Das herkömmliche Modell erlaubt jedoch im Gegensatz zu geschachtelten Mustern weder die Ableitung von Klassen zweiter Ordnung noch die Definition von Klassen höherer Ordnung. Nicht zuletzt erlauben geschachtelte Muster die Beschreibung von Rollen und Unterrollen als Exemplare des sie spielenden Objekts, das ebenfalls durch ein Muster beschrieben werden kann, bzw. der Oberrolle, so daß ein dediziertes Sprachelement zur Beschreibung von Rollen nicht notwendig ist.

5.5 Erweiterung von Exemplaren

Eine weitere Möglichkeit, aus einem Muster ein Exemplar zu erzeugen, ist die Verwendung eines Musters innerhalb der Beschreibung eines Objekts durch einen Objektausdruck. Damit ist eine Erweiterung abstrakt kategorisierter Objekte um weiteres Verhalten möglich. Die Vorgehensweise ist analog zur Ableitung von Mustern.

Dies zeigt Beispiel 5.7. Aus dem Personenmuster der Personenklasse aus Beispiel 5.4 kann eine Person `JerryLewis` exemplifiziert werden, die die besondere Eigenschaft hat, Grimassen schneiden zu können. Im Objektausdruck wird dazu über die `IS`-Klausel das Muster angegeben, das erweitert werden soll (Zeile 1). `JerryLewis` ist damit eine normale Person, die permanent durch `personenklasse` kategorisiert ist, aber den zusätzlichen Befehl `SchneideGrimassen` besitzt (Zeile 5).

5.6 Muster und Delegation

Lieberman [12] schlägt Delegation zum abstrakten Kategorisieren von Objekten vor. Weil sich Delegation immer auf eine konkrete Vorlage, mithin ein anderes Objekt, bezieht, wird zunächst immer das gesamte modellierte Verhalten kategorisiert. Soll im Sinne des abstrakten Kategorisierens nicht das gesamte Verhalten eingeschlossen sein, so muß der nicht kategorisierte Teil durch das delegierende Objekt redefiniert oder gestrichen werden. Entsprechend weisen Dony, Malenfant und Cointe [6] richtig darauf hin, daß etwa entsprechend der Kategorisierung von Verhaltensmustern in den meisten Fällen nur die Komponenten der kategorisierten Objekte und ihre Realisation festgeschrieben werden soll, während ihr Zustand unabhängig voneinander ist. Daher hat ein mithilfe von Delegation erzeugtes Objekt alle zustandstragenden Komponenten selbst zu realisieren. Der Vorteil einer Vorlage ist damit weitgehend verloren. In Self (Ungar und Smith [30]) dienen dagegen *Trait Objects* lediglich zum Faktorisieren gemeinsamer Befehle.

Ein Objekt mit zustandstragenden Komponenten kann seine Befehle an ein solches *Trait Object* delegieren. Andere Objekte nach Vorlage dieses Objekts werden durch Klonieren erzeugt und haben damit zwar einen unabhängigen Zustand, delegieren Befehle aber an dasselbe *Trait Object*. Diese Verwendung der Delegation widerspricht jedoch der Idee des prototypbasierten Ansatzes, in einem System konkrete Objekte als Vorlagen beim Erzeugen anderer Objekte zu verwenden, denn *Trait Objects* sind zustandslos und demnach abstrakt. Sie dienen lediglich als Lager gemeinsamer Befehle und stellen daher eine Optimierung dar, die dem Programmierer nicht sichtbar sein sollte. Keiner der beiden Ansätze nutzt im übrigen eine konkrete Kategorisierung von Objekten, die in ARISTARCH/L durch Klonieren unterstützt wird.

Wenn ein entscheidendes Moment einer Kategorisierung der Zugriff der kategorisierten Objekte auf einen gemeinsamen Zustand ist, so kann ebenfalls Delegation verwendet werden. Lieberman beschreibt beispielsweise eine Menge von Stiften, die immer dieselbe waagrechte Koordinate, aber unabhängige senkrechte Koordinaten besitzen. Dazu verfügen alle Stifte mit einer Ausnahme nur über eine senkrechte Koordinate und delegieren an den Stift, der die gemeinsame waagrechte Koordinate stellt. Man kann sich etwa einen Elektrokardiographen vorstellen, in dem die waagrechte Komponente durch die Position eines Schlittens über einem Papierstreifen bestimmt ist, während die senkrechte Koordinate jedes Stifts sich durch seine Position auf dem Schlitten ergibt. Ein solches Objekt wie diesen Schlitten, das der eigentliche Träger des gemeinsamen Zustands mehrerer Objekte ist, gibt es als Grund für diesen gemeinsamen Zustand immer. Der implizite Zugriff auf den Zustand sollte nach unserer Ansicht aber nicht durch Delegation erreicht werden, denn sein Träger kann nicht als Prototyp, also als Vorlage, der delegierenden Objekte betrachtet werden. In ARISTARCH/L kann es als Klasse der Objekte dienen, denn dort sind auch Klassen abstrakt kategorisierbar. In unserem Beispiel ist der Schlitten mit seiner waagrechten Komponente also Klasse der Stifte, die an dem Schlitten befestigt sind. Geschachtelte Muster erlauben dann die Beschreibung einer Klasse von Schlitten und damit einer Klasse von Elektrokardiographen.

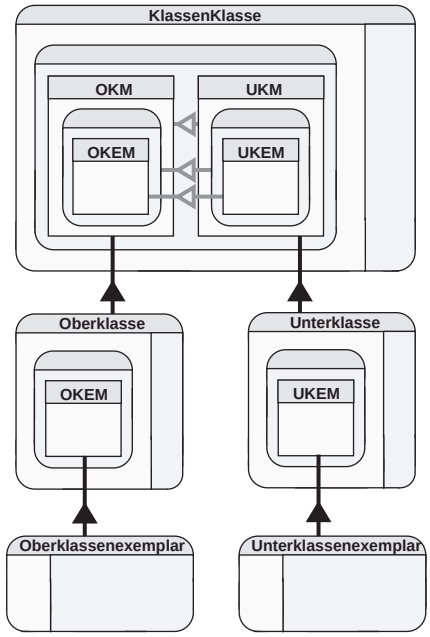

Bild 5.2: Ober- und Unterklassen

Wir betonen, daß Delegation im beschriebenen Modell auch als Vererbung auf der Ebene der Klassen nicht möglich ist. Dies folgt unmittelbar aus unserer Kritik an Delegation und der

Vorstellung von Klassen als Objekte mit Mustern. Anders als in herkömmlichen objektorientierten Sprachen kann demnach eine Klasse nicht von einer anderen, in unseren Begriffen also ein Muster eines Objekts nicht aus Mustern anderer Objekte abgeleitet werden. Die Kategorisierung von Klassen ist aber auch in ARISTARCH/L unproblematisch. Wollen wir eine Klasse beschreiben, deren Exemplare Spezialisierungen der Exemplare einer anderen Klasse sind, so entspricht das der bisherigen Vorstellung einer Ober- und einer Unterklasse. In unserem Modell aber ist entsprechend der konkreten und der abstrakten Kategorisierung neben der festen Bindung auch die Unabhängigkeit zwischen den beiden Klassen möglich. Im ersten Fall werden Ober- und Unterklasse, wie in Bild 5.2 dargestellt, Exemplare einer Klasse mit zwei entsprechend voneinander abgeleiteten Mustern sein. Der Zustand beider Klassen ist dann aber im Gegensatz zu herkömmlichen objektorientierten Sprachen unabhängig voneinander. Der zweite Fall wird dann gegeben sein, wenn ein Programmierer zwar eine andernorts definierte Musterhierarchie wiederverwenden möchte, ihn aber weder spätere Änderungen interessieren noch seine Änderungen auf die bestehende Hierarchie übertragen werden sollen, und er daher die Unterklasse durch Klonieren der Oberklasse erzeugt.

6 Vergleich mit anderen Ansätzen

6.1 Sichten

Hailpern und Nguyen [8] beschreiben ein Modell, in dem die Methode, die nach dem Erteilen eines Auftrags an ein Objekt ausgeführt wird, auch von dem Auftraggeber abhängig ist. Da zudem die Menge der Objekte, die einem Objekt einen bestimmten Auftrag erteilen dürfen, eingeschränkt werden kann, sind in diesem Modell letztlich Sichten mit unterschiedlichen Implementationsversionen für einzelne Komponenten möglich. Der Begriff der Sicht als solcher wird jedoch nicht unterstützt. Zudem fehlt der Bezug auf die benutzte Sicht beim Aufruf von Befehlen aus dem Objekt selbst. Hailpern und Ossher [9] beschreiben zwar ein Modell mit Sichten, aber dieses Modell erlaubt nicht unterschiedliche Implementationsversionen.

Shilling und Sweeney [24] entwickeln ein Modell, das *Views* mit unterschiedlichen Methodenversionen für einzelne Befehle erlaubt. Im Gegensatz dazu gestattet ARISTARCH/L auch unterschiedliche Implementationsversionen von Attributen und Mustern. Im Modell von Shilling und Sweeney können *Views* eigene Attribute besitzen und mehrmals aktiviert werden. Jede Aktivierung kann eigene Variablen für die Attribute der *Views* haben, aber auch Variablen mit anderen Aktivierungen teilen. Das Objekt hat damit eine Beziehung zu den Aktivierungen, die der einer Klasse zu ihren Exemplaren entspricht. *Views* modellieren demnach nicht Aspekte, sondern eigentlich Rollen eines Objekts. Im Modell von Richardson und Schwarz [22] werden ebenfalls Rollen modelliert. *Aspects* sind dort im wesentlichen Objekte mit einem Zeiger auf ein Grundobjekt (engl.: `base`), an das Zugriffe weitergegeben werden, die der *Aspect* nicht selbst realisiert. Es werden nur diejenigen Komponenten des Grundobjekts angeboten, die explizit exportiert worden sind. Die Exportklausel stellt damit lediglich eine syntaktische Kurzform der Realisation einer Komponente des *Aspect* dar, die über den `base`-Zeiger auf die entsprechende Komponente des Grundobjekts zugreift. Pernici [21] beschreibt ein vergleichbares Modell im Bereich der Büroinformationssysteme, das allerdings auf die Spezifikation von Objekten ausgerichtet ist, um vor allem ihre Wiederbenutzung zu unterstützen. Dort werden Rollen in der Tat als *Roles* bezeichnet.

Die Unterscheidung zwischen Aspekten und Rollen eines Objekts machen auch Wieringa, de Jonge und Spruit [31]. Sie beschreiben unterschiedliche Aspekte eines Objekts durch *Dynamic Subclasses*. Im Gegensatz zu herkömmlichen Unterklassen kann ein Objekt zwischen verschiedenen *Dynamic Subclasses* einer bestimmten Dimension wechseln, wobei die Identität

des Objekts erhalten bleibt. Entsprechend kann ein Objekt nicht durch Auswahl einer aktuellen *Dynamic Subclass* zugleich von unterschiedlichen Anwendungen unter unterschiedlichen Aspekten betrachtet werden. Dies ist erst im Zusammenhang mit Rollen möglich. Sie werden durch *Role Classes* beschrieben. Ein Objekt kann eine Rolle entsprechend einer *Role Class* im einem bestimmten Umfang mehrfach spielen. Dabei kann wie bei der Modellierung von Rollen durch Exemplare die Identität des die Rollen spielenden Objekts in den Hintergrund treten. In unserem Modell dient das Objekt selbst als Klasse seiner Rollen und eine Rolle als Klasse ihrer Unterrollen.

Harrison und Ossher [10] argumentieren, daß durch die Betrachtung von Objekten unter unterschiedlichen Aspekten der Begriff des Objekts einen verteilten Charakter erhält. Ihr Ziel ist daher die Entwicklung eines Modells mit den beiden folgenden Eigenschaften. Zum einen sollen Anwendungen, die auf unterschiedlichen Aspekten desselben Objekts beruhen, getrennt voneinander entwickelt werden können, aber dennoch in beliebiger Weise miteinander interagieren. Insbesondere sollen ursprünglich nicht vorgesehene Anwendungen möglich sein, ohne daß bereits vorhandene Objekte unbrauchbar werden und vorhandene Anwendungen geändert oder rekompiliert werden müssen. Zum anderen sollen in allen Anwendungen getrennt die Vorteile von Kapselung, Polymorphismus und Vererbung genutzt werden können. Insbesondere sollen dieselben Objekte in unterschiedlichen Anwendungen entsprechend dem Aspekt, unter dem sie dort betrachtet werden, unabhängig voneinander klassifiziert werden können. Automatisch soll dann der Bezug zwischen den verschiedenen Klassen, die verteilt unterschiedliche Aspekte eines Objekts beschreiben, hergestellt werden, wofür Harrison und Ossher verschiedene Heuristiken vorschlagen. Nach unserer Meinung hat dieser Ansatz einen schwerwiegenden Nachteil, denn damit alle Aspekte eines Objekts letztlich kooperieren, muß der Programmierer einer Anwendung und damit eines Aspekts entweder alle anderen Aspekte vorwegnehmen, um die irgendwann einmal notwendige Kooperation unter den Aspekten durch entsprechende Komponenten zu erreichen, oder im Zuge des Änderns der entsprechenden Klasse die Programmierer aller Anwendungen, die das Objekt unter anderen Aspekten betrachten, davon in Kenntnis setzen, damit diese ihrerseits die entsprechenden Klassen ändern. Die Trennung der verschiedenen Aspekte läßt sich somit im von den Autoren geforderten Maße nicht erreichen. Wir haben uns daher explizit entschieden, die verschiedenen Aspekte zentral durch entsprechende Sichten des Objekts zu beschreiben. ARISTARCH/L erlaubt aber, mehrere Implementationsversionen für eine Komponente in der Abbildung einer Sicht anzugeben, und erreicht so immer eine der Situation angemessene weitgehende Trennung.

FLAVORS [20], COMMONLOOPS [4] und das COMMON LISP OBJECT SYSTEM (CLOS) [5] suchen das Problem des mehrfachen Aufrufs von Methoden eines Befehls – dort einer *Generic Function* – bei mehrfacher Vererbung durch *Method Combinations* zu lösen. CLOS bietet beispielsweise eine *Standard Method Combination*, die zwischen *Primary Methods* und *Auxiliary Methods* unterscheidet. *Auxiliary Methods* werden wiederum in `:before`-, `:after` und `:around`-Methoden eingeteilt. Wird einem Objekt ein Befehl erteilt, so werden erst die `:around`-Methoden ineinander, dann in diesen erst alle `:before`-Methoden, dann die *Primary Methods* und dann die `:after`-Methoden aufgerufen. Dabei werden die Resultate einer Methode an den Vorgänger zurückgegeben, der sie mit seinem Beitrag verknüpfen kann. Die Reihenfolge bestimmt sich durch topologisches Sortieren des Ableitungsgraphen der Klassen. Die *Standard Method Combination* ist allerdings unzureichend, wenn man eine Methode beispielsweise in jedem Fall vor allen anderen eines Befehls ausführen möchte. CLOS bietet dem Programmierer aber die Möglichkeit, andere *Method Combinations* zu definieren. Es hat sich allerdings gezeigt, daß diese Möglichkeit wegen ihrer Komplexität selbst von versierten Programmierern kaum benutzt wird [32]. Wir beschränken uns daher auf einen den *Primary Methods* entsprechenden Ansatz, betrachten aber die Ausgabeparameter als allen Methodenversionen gemeinsame At-

tribute einer Ausführung, in denen durch Seiteneffekte die Resultate entwickelt werden. Snyder [26] kritisiert die Idee der *Method Combinations*, weil bei vermeintlichen Konflikten durch Methoden für denselben Befehl von zwei oder mehr Oberklassen keine Warnung ausgegeben wird, und die Verwendung der Methode einer tatsächlichen Oberklasse – im Gegensatz zu einer Klasse, die lediglich durch den Sortiervorgang als eine solche betrachtet wird – durch eine Klasse nicht garantiert ist. Dieser Einwand ist allerdings unberechtigt, wenn man die Methoden eines Befehls *insgesamt* als Implementation dieses Befehls für die betrachtete Klasse ansieht.

6.2 Delegation und Vererbung

Delegation wird mit eingehenden Beispielen beschrieben von Lieberman [12]. Lieberman erläutert ferner die philosophischen Hintergründe von Delegation und stellt die Behauptung auf, daß Delegation mächtiger sei als Vererbung, da man sie nicht durch Vererbung nachbilden könne. Viele seiner Beispiele werden in späteren Artikeln wieder aufgegriffen, um Probleme mit Delegation zu beschreiben und für alternative Modelle zu argumentieren.

Ein solches alternatives Modell stellt Stein vor [27]. Stein zeigt, daß Liebermans Behauptung falsch und Delegation und Vererbung gleichwertig sind, denn Vererbung ist Delegation auf der Ebene der Klassen. Unser Modell unterscheidet sich hiervon insofern, als wir Delegation und damit Klassen mit einzelnen Mustern als aneinander delegierende Objekte ablehnen. In unserem Modell enthalten Klassen unterschiedliche Muster. Spezialisierung findet auf der Ebene dieser Muster statt. Von Stein stammt auch der Vorschlag, die genaue Strukturgarantie einer Klasse auf eine lediglich *minimale* abzuschwächen.

Im *Treaty of Orlando* [13, 28] einigen sich Lieberman, Stein und Ungar, einer der Entwickler von SELF, darauf, daß in der Tat in verschiedenen Situationen sowohl dynamische als auch statische Elemente zur Beschreibung von Gemeinsamkeiten notwendig sind.

Ungar und Smith [30] beschreiben die prototypbasierte Sprache SELF. SELF verwendet Delegation, um in *Trait Objects* gemeinsame Komponenten von Objekten zu beschreiben, löst aber das Problem des unabhängigen Zustands durch Klonieren eines als Vorlage dienenden Objekts, dem Prototyp. Allerdings geht dadurch die Idee hinter Delegation verloren. Wir verwenden Klonieren eines oder mehrerer Objekte für die konkrete und Muster gleichartiger Objekte für die abstrakte Kategorisierung von Objekten. Die Kombination von *Trait Objects* und Delegation betrachten wir als Optimierung durch das Sprachsystem.

Das Problem der ungerichteten Allgemeinheit von Delegation suchen andere Forscher durch einen regelbasierten Ansatz zu lösen. Minsky und Rozenshtein [18] beschreiben ihre Idee der *Law-Governed Object-Oriented Systems*. In einem solchen System werden Nachrichten zwischen den Objekten durch Regeln verändert, aufgehalten oder an andere als das eigentliche Zielobjekt weitergeleitet. Es unterstützt Delegation nicht direkt, kann sie aber durch entsprechende Regeln nachbilden. Minsky und Rozenshtein [19] zeigen, wie Regeln verwendet werden können, um verschiedene Formen der Benutzung von Delegation in geordnete Bahnen zu weisen. Einen ähnlichen Ansatz verfolgt Almarode [2] mit *Rule-Based Delegation*. Sowohl Minsky und Rozenshtein als auch Almarode allerdings verwenden Delegation in einer Weise, die nicht der Vorstellung von Lieberman entspricht. So werden beispielsweise klassenbasierte Systeme mit Delegation und geeigneten Regeln nachgebildet. Almarode benutzt Delegation sogar dazu, Aufträge innerhalb einer *Part-Of*-Hierarchie weiterzuleiten. In diesem Fall ist das *extended self* von Lieberman nicht notwendig. Delegation wird zum normalen Weiterreichen von Nachrichten, ein Zeichen für eine Form der Benutzung, die nicht der ursprünglichen Idee entspricht. Wir glauben ferner, daß regelbasierte Ansätze für den allgemeinen Einsatz nicht geeignet, weil zu kompliziert in der Anwendung sind. ARISTARCH/L versucht dies durch ein restriktiveres Modell zu umgehen. Regelbasierte Ansätze eignen sich aber sicherlich, um mit Modellideen zu

experimentieren.

Ein den *Law-Governed Object-Oriented Systems* vergleichbares Modell schlagen Akşit, Bergmans und Vural [1] mit den *Composition Filters* vor. *Composition Filters* sind einem Objekt zugeordnet und nehmen Nachrichten an dieses Objekt entgegen. Sie leiten sie unter bestimmten, durch *Filter Elements* gegebenen Bedingungen entweder an das Objekt selbst oder an *Interface Objects* weiter. *Interface Objects* sind durch eine Referenz bestimmt oder befinden sich innerhalb des betrachteten Objekts. Unterschiedliche Aspekte eines Objekts lassen sich in diesem Modell durch ein *Composition Filter* mit einem *Filter Element* je Aspekt darstellen. Der Bezug auf die benutzte Sicht beim Aufruf von Befehlen aus dem Objekt selbst ist nicht herstellbar. Mit Hilfe der Pseudovariablen `server`, die das beauftragte Objekt bestimmt, kann mit durch Referenzen gegebenen *Interface Objects* auch Delegation nachgebildet werden. Entsprechend ist auch die Beschreibung von Sichten durch delegierende Objekte, mithin mit Bezug auf die benutzte Sicht möglich. Insgesamt gelten unsere Argumente gegen Delegation und regelbasierte Ansätze auch im Falle des Modells der *Composition Filters*.

Dony, Malenfant und Cointe [6] suchen durch eine Taxonomie prototypbasierter Sprachen eine ideale Sprache zu entwickeln. Dabei beschreiben sie verschiedene Probleme von Delegation wie das des autonomen Zustands delegierender Objekte oder die mangelhafte Unterstützung von *Split Objects*. Sie stellen ferner fest, daß die Beschreibung gemeinsamer Komponenten mit einer Strukturgarantie für eine Menge von Objekten, wie sie mit Klassen möglich ist, weder durch Klonieren noch durch Delegation erreicht werden kann. Wir unterstützen *Split Objects* durch Sichten und impliziten Zugriff von Exemplaren auf Komponenten ihrer Klasse. Die Notwendigkeit autonomer Objekte einerseits und der Strukturgarantie für eine Menge von Objekten andererseits führt dazu, daß ARISTARCH/L sowohl Klonieren als auch Muster in Objekten anbietet. Delegation wird von ARISTARCH/L nicht unterstützt, da wir sie für einen Mechanismus zur Implementation von Sprachelementen halten, der auf der Ebene der Sprache nicht sichtbar sein sollte.

7 Zusammenfassung

Wir haben ein Objektmodell für ein System mit einem einheitlichen Objektraum vorgestellt, daß den besonderen Forderungen eines solchen Systems gerecht wird. Das Modell erlaubt sowohl die direkte Beschreibung von Objekten ohne Klassen über *Objektausdrücke*, als auch die Veränderung bestehender Objekte, um vorhandene Informationen zu erhalten. Verschiedene Aspekte eines Objekts können mittels *Sichten* und *Versionen* getrennt voneinander beschrieben werden. Das Modell berücksichtigt die konkrete und die abstrakte Kategorisierung sowie die Erweiterung abstrakt kategorisierter Objekte mittels *Klonieren*, *Exemplarmustern* und Objektausdrücken, die auf Muster zurückgreifen können. Muster erlauben ferner die Beschreibung der unterschiedlichen Rollen, die ein Objekt spielt. Nicht zuletzt in diesem Zusammenhang sind *geschachtelte Exemplarmuster* notwendig. Sie erlauben die Realisierung von Klassen, die ihren Exemplaren dasselbe Verhaltensmuster vorgeben, sich aber in ihrem Zustand unterscheiden. Damit stehen erweiterte Möglichkeiten zur Beschreibung von Gemeinsamkeiten zwischen Objekten zur Verfügung, ohne Delegation zwischen Objekten oder Vererbung zwischen Klassen auf der Ebene der Sprache zu verwenden.

Literatur

[1] M. AKŞIT, L. BERGMANS UND S. VURAL. An Object-Oriented Language-Database Integration Model: The Composition-Filters Approach. In: Madsen [14], S. 372–395.

[2] J. Almarode. Rule-Based Delegation for Prototypes. In: Meyrowitz [17], S. 363–370.

[3] A. P. Black, N. C. Hutchinson, E. Jul, H. M. Levy und L. Carter. Distribution and Abstract Types in Emerald. *IEEE Transactions on Software Engineering*, **SE-13**(1):65–76, Januar 1987.

[4] D. G. Bobrow, K. Kahn, G. Kiczales, L. Masinter, M. Stefik und F. Zdybel. CommonLoops: Merging Lisp and Object-Oriented Programming. In: Meyrowitz [15], S. 17–29.

[5] L. G. DeMichiel und R. P. Gabriel. The Common Lisp Object System: An Overview. In: J. Bézivin, J.-M. Hullot, P. Cointe und H. Lieberman, Hrsg., *Proceedings of the European Conference on Object-Oriented Programming* (Paris, Frankreich, 15.–17. Juni 1987), *Lecture Notes in Computer Science*, Bd. 276, S. 151–170. Springer-Verlag, Berlin, 1987.

[6] C. Dony, J. Malenfant und P. Cointe. Prototype-Based Languages: From a New Taxonomy to Constructive Proposals and Their Validation. In: A. Paepcke, Hrsg., *Proceedings of Object-Oriented Programming Systems, Languages, and Applications* (Vancouver, British Columbia, Kanada, 18.–22. Oktober 1992), *ACM SIGPLAN Notices*, **27**(10):201–217, Oktober 1992. ACM.

[7] M. A. Ellis und B. Stroustrup. *The Annotated* C++ *Reference Manual.* Addison-Wesley, Reading, Massachusetts, 1990.

[8] B. Hailpern und V. Nguyen. A Model for Object-Based Inheritance. In: Shriver und Wegner [25], S. 147–164.

[9] B. Hailpern und H. Ossher. Extending Objects to Support Multiple Interfaces and Access Control. *IEEE Transactions on Software Engineering*, **16**(11):1247–1257, November 1990.

[10] W. Harrison und H. Ossher. Subject-Oriented Programming (A Critique of Pure Objects). In: A. Paepcke, Hrsg., *Proceedings of Object-Oriented Programming Systems, Languages, and Applications* (Washington, Washington, D.C., 26. September–1. Oktober 1993), *ACM SIGPLAN Notices*, **28**(10):411–428, Oktober 1993. ACM.

[11] W. R. LaLonde, D. A. Thomas und J. R. Pugh. An Exemplar Based Smalltalk. In: Meyrowitz [15], S. 322–330.

[12] H. Lieberman. Using Prototypical Objects to Implement Shared Behaviour in Object-Oriented Systems. In: Meyrowitz [15], S. 214–223.

[13] H. Lieberman, L. A. Stein und D. M. Ungar. Of Types and Prototypes: The Treaty of Orlando. In: L. Power und Z. Weiss, Hrsg., *Addendum to the Proceedings of Object-Oriented Programming Systems, Languages, and Applications* (Orlando, Florida, 4.–8. Oktober 1987), *ACM SIGPLAN Notices*, **23**(5):43–44, Mai 1988. ACM.

[14] O. L. Madsen, Hrsg. *Proceedings of the European Conference on Object-Oriented Programming* (Utrecht, Niederlande, 29. Juni–3. Juli 1992), *Lecture Notes in Computer Science*, Bd. 615. Springer-Verlag, Berlin, 1992.

[15] N. Meyrowitz, Hrsg. *Proceedings of Object-Oriented Programming Systems, Languages, and Applications* (Portland, Oregon, 29. September–2. Oktober 1986), *ACM SIGPLAN Notices*, **21**(11), November 1986. ACM.

[16] N. Meyrowitz, Hrsg. *Proceedings of Object-Oriented Programming Systems, Languages, and Applications* (Orlando, Florida, 4.–8. Oktober 1987), *ACM SIGPLAN Notices*, **22**(12), Dezember 1987. ACM.

[17] N. Meyrowitz, Hrsg. *Proceedings of Object-Oriented Programming Systems, Languages, and Applications* (New Orleans, Louisiana, 1.–6. Oktober 1989), *ACM SIGPLAN Notices*, **24**(10), Oktober 1989. ACM.

[18] N. H. Minsky und D. Rozenshtein. A Law-Based Approach to Object-Oriented Programming. In: Meyrowitz [16], S. 482–493.

[19] N. H. Minsky und D. Rozenshtein. Controllable Delegation: An Exercise in Law-Governed Systems. In: Meyrowitz [17], S. 371–380.

[20] D. A. Moon. Object-Oriented Programming with Flavors. In: Meyrowitz [15], S. 1–8.

[21] B. Pernici. Objects with Roles. In: F. H. Lochovsky und R. B. Allen, Hrsg., *Proceedings of the Conference on Office Information Systems* (Cambridge, Massachusetts, 25.–27. April 1990), *ACM SIGOIS Bulletin*, **11**(2/3):205–215, April/Juli 1990. ACM.

[22] J. Richardson und P. Schwarz. Aspects: Extending Objects to Support Multiple, Independent Roles. In: J. Clifford und R. King, Hrsg., *Proceedings of the International Conference on Management of Data* (Denver, Colorado, 29.–31. Mai 1991), *ACM SIGMOD Record*, **20**(2):298–307, Juni 1991. ACM.

[23] C. Schaffert, T. Cooper, B. Bullis, M. Kilian und C. Wilpolt. An Introduction to Trellis/Owl. In: Meyrowitz [15], S. 9–16.

[24] J. J. Shilling und P. F. Sweeney. Three Steps to Views: Extending the Object-Oriented Paradigm. In: Meyrowitz [17], S. 353–361.

[25] B. Shriver und P. Wegner, Hrsg. *Research Directions in Object-Oriented Programming*. Series in Computer Systems. MIT Press, Cambridge, Massachusetts, 1987.

[26] A. Snyder. Inheritance and the Development of Encapsulated Software Systems. In: Shriver und Wegner [25], S. 165–188.

[27] L. A. Stein. Delegation is Inheritance. In: Meyrowitz [16], S. 138–146.

[28] L. A. Stein, H. Lieberman und D. M. Ungar. A Shared View of Sharing: The Treaty of Orlando. In: W. Kim und F. H. Lochovsky, Hrsg., *Object-Oriented Concepts, Databases, and Applications*, ACM Press Frontier Series, S. 31–48. Addison-Wesley, Reading, Massachusetts, 1989.

[29] C. A. Szyperski. Import Is Not Inheritance – Why We Need Both: Modules and Classes. In: Madsen [14], S. 19–32.

[30] D. M. Ungar und R. B. Smith. Self: The Power of Simplicity. In: Meyrowitz [16], S. 227–242.

[31] R. Wieringa, W. d. Jonge und P. Spruit. Roles and Dynamic Subclasses: A Modal Logic Approach. In: M. Tokoro und R. Pareschi, Hrsg., *Proceedings of the European Conference on Object-Oriented Programming* (Bologna, Italy, 4.–8. Juli 1994), *Lecture Notes in Computer Science*, Bd. 821, S. 32–59. Springer-Verlag, Berlin, 1994.

[32] P. H. Winston und B. K. P. Horn. Lisp, S. 509. Addison-Wesley, Reading, Massachusetts, dritte Auflage, 1989.