ODABA 9.0

– Real Objects –

run Software-Werkstatt GmbH
Weigandufer 45
12059 Berlin

Tel: +49 (30) 609 853 44
e-mail: run@run-software.com
web: www.run-software.com

Berlin, October 2012

Content

Kapitel 2
Die Sprachwelt

In diesem Kapitel soll der Zusammenhang zwischen Datenmodell und Sprachmodell, der Die Grundlage der ODABA2-Philosophie bildet, dargestellt werden. Es geht um die Zusammenhänge zwischen der realen Welt, der Sprache und den Datenmodellen. Datenmodelle vermitteln gleich der Sprache Bilder der Wirklichkeit. Dies ist nicht die einzige Gemeinsamkeit, die Sprache und Datenmodell verbindet. Wie wir sehen werden, spielt die Sprache zum einen im Abbildungsprozeß selbst eine Rolle, zum anderen werden Prinzipien der Sprache zunehmend in Datenmodellen umgesetzt, so daß die Datenmodelle der Sprache immer „ähnlicher“ werden.

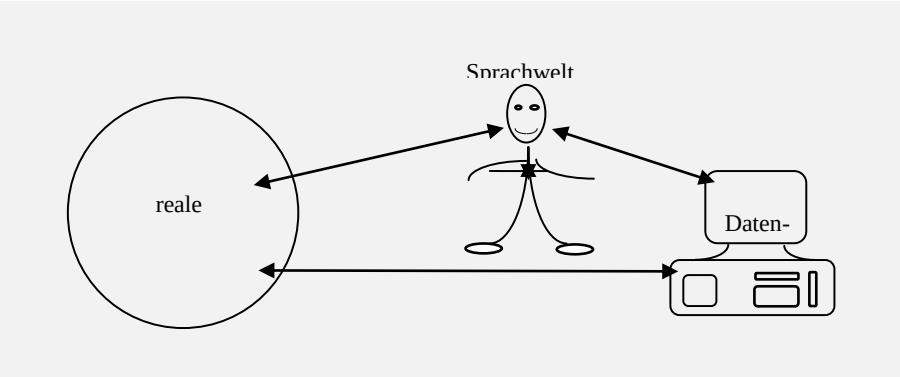


Abbildung 3.1: Abbildung der realen Welt auf dem Computer

Bei sprachlichen Darstellungen geht es im wesentlichen um die Darstellung von Sachverhalten. Dabei wird schnell klar, daß reale Objekte spezielle Sachverhalte sind, die sich von anderen Sachverhalten durch ihre reale, vom Bewußtsein unabhängige Existenz unterscheiden. Da eben diesen Objekten eine wesentliche Rolle zukommt, scheint es dennoch berechtigt, sie in den Mittelpunkt des Modells

zu stellen. Für die Darstellung von realen Objekten finden sich viele wertvolle Hinweise in der Sprache.

Verfälscht Sprache die Wirklichkeit?

Datenmodelle vermitteln gleich der Sprache Abbilder der Wirklichkeit. Durch Datenmodelle kann ein Teil dessen, was die Sprache ausdrückt, genauer dargestellt werden. Sprache ist im Gegensatz zu Datenmodellen jedoch unschärfer und läßt vielfältige Deutungen zu. Es ist offensichtlich, daß es der Sprache nicht gelingt, das vollständige Bild eines realen Sachverhaltes zu vermitteln. Sprache wird immer auswählen und weglassen. Es ist auch gar nicht Zweck der Sprache, ein detailgetreues Abbild der Wirklichkeit zu erzeugen. Durch die Sprache findet eine **systematische Abbildung** der realen Welt in die Sprachwelt statt. Während die Welt höchstens einer natürlichen Ordnung folgt, stellt die Sprache die Sachverhalte entsprechend verschiedenen menschlichen Systematiken dar. Die Sprache ordnet die Dinge der Welt in menschlichem Sinne. Durch diese Ordnung erfahren die Sachverhalte in der Sprache eine Wertung. Diese Wertung beginnt bereits bei dem Begriff, mit dem Sachverhalte eingeordnet werden.

Durch solche Begriffe wird ein Sachverhalt einer Art zugeordnet. Die **Systematik der Arten** ist eines der wesentlichen Prinzipien, auf denen die Systematik sprachlicher Darstellungen beruht. Indem die Sprache Sachverhalte systematisiert, verleiht sie dem Abbild eine neue Qualität. Sie ordnet die Sachverhalte bestimmten Interessen oder Vorstellungen unter und widerspiegelt somit die Interessen und Vorstellungen des Darstellenden. Zusammenhänge, die in der realen Welt kaum sichtbar werden, können somit durch Sprache auf einfache Weise dargestellt werden.

Wie aus der Sprachwelt die Datenwelt entsteht

Die durch Wertungen angereicherten Sprachbilder werden im zweiten Abbildungsprozeß als Daten im Computer (Datenwelt) dargestellt. Während die Sprache jedem Einzelnen noch einen Spielraum dafür läßt, die Begriffe zu deuten, sind Begriffe in der Datenwelt genau definiert.

Welche Merkmale zu einem Sachverhalt erfaßt werden, hängt nicht nur davon ab, was der Sachverhalt ist, sondern was an ihm in einem bestimmten Zusammenhang wichtig ist, also von der Speziellen Sicht auf diesen Sachverhalt. Die Wertung eines Sachverhaltes drückt sich somit nicht nur in der Art aus, der dieser Sachverhalt durch den Begriff zugeordnet wird, sondern auch in den Merkmalen, die zu dem Sachverhalt abgebildet werden. Erst dadurch, daß Sachverhalte in gleicher Weise in der Datenwelt dargestellt werden, werden sie vergleichbar. Eben diese Vergleichbarkeit ist ein Ziel der Darstellung von Sachverhalten in der Datenwelt. Dennoch folgen Datenmodelle den gleichen Prinzipien der systematischen Abbildung wie die Sprache. In diesem Sinn übernimmt das Datenmodell die Wertungen der Sprachwelt und stellt diese in der Datenwelt dar.

Was ist ein Modell?

Die Sprachwelt und die Datenwelt sind Abbilder der realen Welt. Diesen Abbildungen liegen bestimmte Prinzipien zugrunde, denen der Abbildungsprozeß folgt. Da die Sprache eine systematische Abbildung ist, finden diese Prinzipien nicht in der Syntax, sondern in der Systematik der Sprache ihren Ausdruck. Die Systematik der Sprache beschreibt, wie Sachverhalte darzustellen sind. Eine solche Ordnung, die die Regeln zur Beschreibung eines Sachverhaltes definiert, wird als **Schema** bezeichnet. Auch die Beschreibung eines Schemas erfolgt nach bestimmten Regeln. Diese sind im **Modell** definiert. Das Modell ist also ein Schema für die Schemadefinition, ist die Beschreibung einer Beschreibung.

Die Prinzipien sprachlicher Darstellung zu definieren heißt, das Sprachmodell zu beschreiben. Das Modell der menschlichen Sprache ist jedoch so komplex, daß es gegenwärtig nicht vollständig im Datenmodell dargestellt werden kann. ODABA2 bedient sich deshalb eines **reduzierten Sprachmodells**.

Die Orientierung an Sprachmodellen ist mit zwei wesentlichen Vorteilen verbunden. Zum einen erleichtert sie das Verständnis des Modells, da Sprache etwas ist, was jeder kennt. Die Sprache hat sich über Jahrtausende hinweg als zweckmäßiges Abbildungsmodell erwiesen, das bisher in seinen Prinzipien nicht übertroffen wurde. Zum anderen ist es durch die Definition eines reduzierten Sprachmodells, das auf das Datenmodell bezogen ist möglich, eine natürliche Sprachschnittstelle (**Natural Language Interface**) zu definieren.

2.1 Das Modell

Um das Sprachmodell und das objektorientierte Modell in Beziehung zueinander setzen zu können, soll der Modellbegriff nun etwas genauer gefaßt werden. Wie Sachverhalte der realen Welt in der Sprache darzustellen sind, wird durch die Systematik der Sprache festgelegt. Die Systematik der Sprache findet ihren Ausdruck in einem Schema. Auch ein Schema wird nach bestimmten Regeln beschrieben. Um dieses Schema zu beschreiben, müssen die Sachverhalte der Schemaebene definiert werden. Die Regeln, die dieser Definition eines Schemas zugrunde liegen, werden im Modell definiert. In diesem Sinne kennt ODABA2 drei Abbildungsebenen. Während die Darstellung eines bestimmten Sachverhaltes der realen Welt auf der Sachebene angesiedelt ist, wird das Schema, die Beschreibung der Darstellung, auf einer Metaebene - der Schemaebene - dargestellt. Die Darstellung der Regeln der Schemaebene, also die Beschreibung der Beschreibung erfolgt wiederum auf einer Metaebene der Schemaebene - auf der Modellebene.

- Sachebene

  Auf der Sachebene werden Sachverhalte der realen Welt dargestellt. Diese werden nach einem bestimmten Schema beschrieben.

- Schemaebene

  Auf der Schemaebene wird das Schema beschrieben, das den Beschreibungen auf der Sachebene zugrunde liegt.

- Modellebene

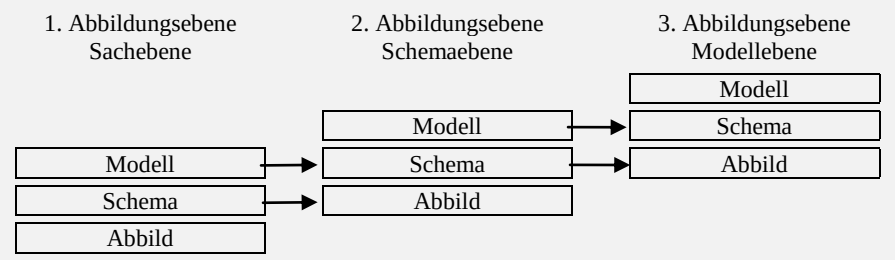
  Auf der Modellebene wird beschrieben, nach welchen Prinzipien ein Schema definiert wird.

Wandlungen beim Wechsel der Ebenen

Aussagen auf der Sachebene basieren auf Begriffen, die eine bestimmte Bedeutung haben. Die Begriffe und ihre Bedeutung existieren unabhängig vom realen Sachverhalt, d.h. der Begriff selbst ist ein abstrakter Sachverhalt. Abstrakte

Sachverhalte werden auf der Schemaebene beschrieben. Begriffe wie Merkmal, Art, Objekt, Kategorie sind Sachverhalte der Schemaebene (abstrakte Sachverhalte), mit deren Hilfe die Darstellung konkreter Sachverhalte beschrieben werden kann. Indem abstrakte Sachverhalte auf der Schemaebene beschrieben werden, werden sie dort zu konkreten Sachverhalten.

Jede Ebene ist also durch die konkreten Sachverhalte, die sie darstellt, und durch die abstrakten Sachverhalte, die das Schema dieser Ebene definieren, bestimmt. Dabei werden abstrakten Sachverhalte auf der nächsthöheren Ebene zu konkreten Sachverhalten. Die Sicht auf die Ebenen basiert also immer auf einem bestimmten Standpunkt. Da auch auf den Metaebenen Sachverhalte entstehen, die in geeigneter Weise beschrieben werden müssen, ergibt sich ausgehend von jeder Ebene jeweils wieder eine Sach-, Schema- und Modellebene, nur eben um eine Stufe nach oben verschoben. Somit drücken die Ebenen nicht die tatsächlichen Darstellungsebenen aus, sondern ihre Stellung bei der Beschreibung der Sachverhalte einer beliebigen Ebene.



Abbildungsebenen

Auf jeder Ebene werden die Sachverhalte der entsprechenden Ebene abgebildet. Da das Schema auf der Schemaebene zum Gegenstand der Abbildung wird, ist es hier Abbild. In der zweiten Abbildungsebene werden also die Sachverhalte des Schemas abgebildet. Das Modell wird in der zweiten Abbildungsebene zum Schema, das die Systematik beschreibt, nach der die Schemata dargestellt werden. Zwischen den Ebenen besteht jeweils eine 1:N-Beziehung. Zu einem Modell kann es beliebig viele Schemata geben, für ein Schema können beliebig viele Abbildungen auf der Sachebene erzeugt werden.

Rekursive Modelle

Da sich auf jeder Abbildungsebene eine neue Modellebene ergibt, stellt sich die Frage, ob die Anzahl der Ebenen begrenzt werden kann. Ist das Schema der zweiten Abbildungsebene (Schemaebene) identisch mit dem Schema der ersten Abbildungsebene (Sachebene), sprechen wir von einem rekursiven Schema. Sprachmodelle besitzen i.allg. ein rekursives Schema, d.h. es wird auf allen Darstellungsebenen das gleiche Schema verwendet. Damit stellt jede Spracherweiterung eine Schemaerweiterung dar, was zur maximalen Flexibilität des Modells führt. Datenmodelle mit rekursivem Schema sind derzeit jedoch noch nicht bekannt. Derzeit bilden sich jedoch rekursive Datenmodelle heraus, die wesentlich flexibler sind, als die bisherigen nicht rekursiven Modelle (wie z.B. alle relationalen Datenmodelle). Von rekursiven Modellen sprechen wir, wenn die Schemata der zweiten und dritten Abbildungsebene identisch sind. In diesem Fall ist ein Modell ausreichend, um beliebige Ebenen zu beschreiben.

Ausdrücken auf mehreren Ebenen

Alle hier definierten Modelle gehen von der Systematik der Arten aus. Sachverhalte werden nach bestimmten Kriterien Arten zugeordnet und innerhalb der Arten durch bestimmte Merkmale beschrieben. Die Beschreibung eines Sachverhaltes erfolgt also, indem er einer Art zugeordnet wird, seine Merkmale benannt und mit entsprechenden Werten belegt werden.

Da die Beispiele auf der Sachebene angesiedelt sind, werden wir in ihnen Aussagen der Schemaebene verschlüsselt darstellen, um den Text etwas zu entlasten. Besonders wichtig ist in den verschiedenen Zusammenhängen die Unterscheidung zwischen Arten, Merkmalen und Werten (Zuständen):

PERSON — Durch Kapitälchen werden Begriffe gekennzeichnet, die Arten darstellen. Diese Angabe entspricht also der Metaaussage: „*Person* ist eine ART“.

Größe Fett gedruckte Begriffe bezeichnen in den Beispielen Merkmale einer Art. Die Art, der die Merkmale angehören, geht aus dem jeweiligen Zusammenhang hervor. Fettdruck in Beispielen drückt also auf der Schemaebene aus: *"Größe* ist ein MERKMAL".

Anton Müller Kursiv gedruckte Satzteile stellen einen Zustand dar. Sie identifizieren einen konkreten Sachverhalt (Objekt) oder drücken einen bestimmten Wert aus. Auf der Schemaebene bedeutet Kursivschrift also: „*Anton Müller* ist ein ZUSTAND."

2.2 Die Prinzipien des Sprachmodells

Ein wesentlicher Teil der Sprache beschreibt Sachverhalte der realen Welt. Diese eingeschränkte Sprache wird als Sachsprache bezeichnet. Die Sachsprache beschränkt sich darauf, Sachverhalte unterschiedlichster Art zu beschreiben. Sie beschreibt Gegenstände genauso wie Tätigkeiten oder abstrakte Zusammenhänge. Selbst Ideen oder Gesetze und Regeln sind Sachverhalte, auch wenn sie nicht materieller Natur sind. Der Sachverhalt ist in der Sachsprache der Begriff höchster Abstraktion. Die Darstellung von Sachverhalten besteht dabei aus der Sachbeschreibung und der Beschreibung ihres Verhaltens.

Da die Sachsprache eine systematische Abbildung ist, legt sie der Darstellung der Sachverhalte eine Systematik zugrunde. Diese Systematik kommt vor allem im Prinzip der Arten zum Ausdruck. Zur Zeit sind drei Prinzipien bekannt, auf denen die Systematik sprachlicher Darstellungen beruhen.

- Prinzip der Arten

  Sachverhalte werden über Begriffe bestimmten Arten zugeordnet. Diese bilden über ein System von Ober- und Unterbegriffen die Hierarchie der Arten.

- Kausalitätsprinzip

  Durch das Kausalitätsprinzip werden Zusammenhänge zwischen Ursachen und ihren Wirkungen beschrieben.

- Temporalprinzip

  Das Temporalprinzip stellt Sachverhalte in einen zeitlichen Zusammenhang. Dadurch wird es möglich, Sachverhalte in ihrer zeitlichen Entwicklung darzustellen.

Im ODABA2-Datenmodell liegt der Schwerpunkt zur Zeit auf dem Prinzip der Arten. Das Kausalitätsprinzip wird in einer ersten Version unterstützt, ist aber noch sehr unvollständig. Das Temporalprinzip spielt nur eine untergeordnete Rolle (z.B. bei der Darstellung von Structure-Versionen). Im weiteren werden Methoden vorgestellt, die es erlauben, Zusammenhang so durch natürliche Sprache zu beschreiben, daß aus dieser Beschreibung ein widerspruchsfreies Objektmodell erzeugt werden kann.

2.3 Das artbezogene Sprachmodell

Ein einfachen Modell ist das artbezogene Sprachmodell, daß sich auf ein **einfaches Prinzip der Arten** beschränkt. Das einfache Prinzip der Arten beschränkt sich auf Zustandsbeschreibungen, d.h. es wird kein Verhalten dargestellt. Das einfache Prinzip der Arten beruht darauf, Sachverhalte einer Art zuzuordnen und aus der Sicht dieser Art darzustellen.

Arten von Sachverhalten drücken in vielen Zusammenhängen die Sicht eines Subjektes auf einen Sachverhalt aus, indem sie die aus einer bestimmten Sicht wesentlichen Merkmale eines Sachverhaltes hervorheben. Obwohl es möglich ist, verschiedene Sichten auf einen Sachverhalt durch unterschiedliche Arten darzustellen, begnügen sich einfache Darstellung mit der Tatsache, daß ein Sachverhalt genau einer Art zugeordnet wird. So wird z.B. in der Biologie jedes Lebewesen einer Art zugeordnet, durch welche die wesentlichen Merkmale dieses Lebewesens charakterisiert werden. Das einfache Prinzip der Arten besteht nun gerade darin, jeden Sachverhalt genau einer Art zuzuordnen und aus der Sicht dieser Art darzustellen.

Das artbezogene Sprachmodell legt bezüglich einer Art auch die **Merkmale** fest, die für einen Sachverhalt dargestellt werden sollen. Indem eine Art dadurch definiert wird, daß Sachverhalte dieser Art durch bestimmte Merkmale dargestellt werden, wird eine bestimmte Sicht auf diese Sachverhalte festgelegt. Der Sachverhalt ist genau bezüglich der festgelegten Merkmale zu betrachten, da diese die entscheidenden Seiten des Sachverhaltes darstellen. Aus diesem Grunde werden die Arten, die durch festgelegte Merkmale beschrieben werden, auch als **Sicht** bezeichnet. Merkmale sind in dem Sinne kontextbezogene Sachverhalte, als daß sie im Zusammenhang, im Kontext einer Sicht definiert werden. Somit sind auch die Begriffe, die zur Bezeichnung dieser Merkmale verwendet werden, nur im Rahmen einer Sicht eindeutig.

Der Schlüssel zum Sachverhalt

Um die Sicht auf einen Sachverhalt, sein Abbild, mit dem real existierenden Sachverhalt in Verbindung bringen zu können, muß dieser in geeigneter Weise identifiziert werden. Die Identifikation eines Sachverhaltes erfolgt durch artbezogene **Schlüssel**. Ein Schlüssel faßt ein oder mehrere Merkmale der Sicht zusammen, die in einem bestimmten Zusammenhang geeignet sind, einen Sachverhalt zu identifizieren.

Schlüssel sind eingeschränkte Sichten, die sich im Rahmen einer Sicht auf die Merkmale beziehen, die zur Identifikation des Sachverhaltes dieser Art in einem bestimmten Zusammenhang erforderlich sind. Für eine Sicht können ein oder mehrere Schlüssel definiert sein, mit deren Hilfe Sachverhalte in verschiedenen Zusammenhängen identifiziert werden können.

Die einfache Systematik der Arten

Die Darstellung der Sachverhalte erfolgt je nach Komplexität der geforderten Darstellung in unterschiedlicher Weise. Dabei zeigt sich, daß es neben den Sichten zwei weitere Kategorien von Arten gibt, die wesentliche Unterschiede hinsichtlich der Darstellung der Art aufweisen.

- Sichten

  Komplexe Sachverhalte werden durch Sichten dargestellt, die durch ein oder mehrere Merkmale definiert werden. Die Merkmale einer Sicht beschreiben untergeordnete Sachverhalte. Auch Merkmale als untergeordnete Sachverhalte werden in definierter Weise dargestellt. Das erfolgt wieder entsprechend einer Art. Ist ein untergeordneter Sachverhalt komplexer Natur (wie z.B. die Anschrift im Kontext einer Person), kann diese Art wieder in einer Sicht bestehen, die beschreibt, wie Anschriften darzustellen sind.

- Systematiken

  Systematiken werden gebildet, um Sachverhalte in Gruppen zu gliedern, die als Kategorien bezeichnet werden. Eine Systematik ist durch eine begrenzte Anzahl von Kategorien definiert. Dabei wird jeder Sachverhalt im Rahmen einer Systematik genau einer Kategorie zugeordnet. Systematiken werden als Arten für Merkmale verwendet, wenn über dieses Merkmal die Zuordnung zu einer Kategorie ausgedrückt werden soll.

- Elementare Arten

  Elementare Arten bilden die Grundlage des Sprachmodells. Sie werden als bekannt vorausgesetzt. Jede Definition eines Merkmals kann auf elementare Arten Bezug nehmen, ohne daß sie weiter definiert werden müssen. Elementare Arten sind somit das Ende jeder Sichtdefinition. **Zahlen** zur Darstellung quantitative Größen dargestellt oder **Bezeichner**, die der Identifikation von Sachverhalten dienen sind Beispiele für elementare Arten.

Die Schemadefinition im artbezogenen Sprachmodell

Die Schemadefinition im artbezogenen Sprachmodell besteht in der Definition von Arten, mit deren Hilfe Sachverhalte, die einer Art zugeordnet wurden, dargestellt werden können. Die Definition einer Art erfolgt durch Festlegen der Kategorie der Art. Da elementare Arten als definiert angesehen werden, können auf der Schemaebene nur Systematiken und Sichten definiert werden. Sichten werden definiert, indem die Merkmale bestimmt werden, die im Rahmen einer Sicht

beschrieben werden müssen. Systematiken werden durch die Aufzählung der Kategorien der Systematik festgelegt.

Das artbezogene Sprachmodell ist also im wesentlichen durch die Begriffe ART, SICHT, SYSTEMATIK, ELEMENTARE ART, MERKMAL und SCHLÜSSEL definiert. Jeder Sachverhalt der Schemaebene läßt sich einem dieser Modellbegriffe (Arten der Modellebene) zuordnen. Die auf der Schemaebene bekannten Arten werden in der Definition des Sprachmodells zusammengefaßt. Diese Zusammenfassung erfolgt in Form eines Lexikons, dessen Aufbau im folgenden beschrieben wird.

Die Mängel des artbezogenen Sprachmodells

Die konzeptionellen Mängel des artbezogenen Sprachmodells sind offensichtlich. Im Rahmen dieses Sprachmodells sind nur Zustandsbeschreibungen von Sachverhalten möglich. Nicht einmal die Hierarchien der Arten, die durch Ober- und Unterbegriffe gebildet werden, können in hier dargestellt werden. Die Beschreibung der Sachverhalte reduziert sich auf reine Zustandsbeschreibungen, ohne daß Beziehungen zwischen verschiedenen Sachverhalten im Modell dargestellt werden können.

Ein weiterer Mangel liegt in der Beschränkungen auf das Prinzip der Arten. Es können weder kausale noch temporale Zusammenhänge dargestellt werden. Praktisch ist jedoch keine Anwendung denkbar, die nicht in irgendeiner Weise kausale Zusammenhänge abbilden muß.

2.4 Das objektorientierte Sprachmodell

Dieser Abschnitt beschäftigt sich mit der Frage, was Objekte wirklich sind und wie sie in der ODABA2-Sprachwelt dargestellt werden. Da das ODABA2-Sprachmodell auf allgemeinen Erfahrungen aufbaut, soll es im folgenden allgemein als **objektorientiertes Sprachmodell** (**OSM**) bezeichnet werden. Hinsichtlich der Feinheit und der Vollständigkeit läßt auch das hier vorgestellte OSM viele Fragen offen. Weitere Annäherungen an die Sprachwelt können jedoch durch neue Modelle oder dadurch erreicht werden, daß das OSM verfeinert oder vervollständigt wird.

Im OSM findet zum einen das Kausalitätsprinzip seinen Platz. Zum anderen wird das Prinzip der Arten verfeinert, indem vielfältige Beziehungen zwischen den Arten hergestellt werden. Somit stellt das OSM eine Erweiterung bzw. Verfeinerung des artbezogenen Sprachmodells dar. Gleichzeitig wird offenbar, daß dieses Prinzip der Verfeinerung weitergeführt werden kann und zu neuen Modellen oder Erweiterungen des OSM führen wird.

2.4.1 Das Prinzip der Arten

Das Prinzip der Arten besteht zum einen in der detaillierteren Darstellung der Sachverhalte durch ihren Zustand sowie durch die Darstellung von Beziehungen und Verhalten. Dabei werden verschiedene Formen von Beziehungen zwischen Sachverhalten dargestellt, die im folgenden genauer beschrieben werden.

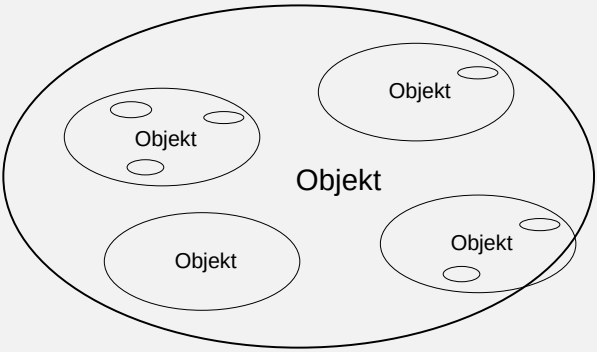
2.4.1.1 Darstellung allgemeiner Sachverhalte

Was die Sachsprache auf jeder Ebene abbildet, sind Sachverhalte. **Sachverhalte** sind Gegebenheiten der realen Welt. Sie können sich auf Dinge, Zusammenhänge, Prozesse, Ideen und anderes beziehen. Alles, was die Sachsprache beschreibt, sind Sachverhalte. Sachverhalte können konkret oder abstrakt sein. Konkrete Sachverhalte sind identifizierbare Dinge oder Erscheinungen der realen Welt.

Konkrete Sachverhalte werden auf der Sachebene beschrieben. Abstrakte Sachverhalte beschreiben das Schema einer Darstellung und werden auf der Schemaebene abgebildet.

Das Objekt – ein konkreter Sachverhalt

Ein Sachverhalt kann eine real existierende Gegebenheit sein. In diesem Fall handelt es sich um ein Objekt. Objekte sind konkrete Sachverhalte, die eine zeitlich und räumlich begrenzte Existenz haben und unabhängig von anderen Sachverhalten existieren. Objekte sind meistens dinglicher Art. Ein Objekt kann selbst aus mehreren Objekten zusammengesetzt sein, die ihrerseits wieder aus Objekten bestehen können. Diese untergeordneten Objekte eines Objektes werden als Teile bezeichnet.



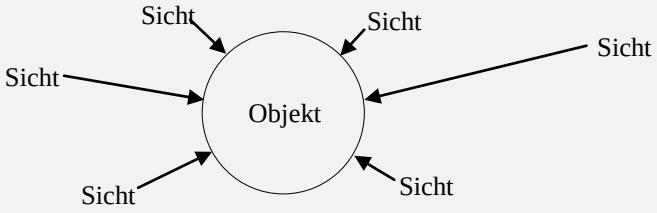
Innen und Außen der Objekte

Während ein Objekt nach außen als geschlossene Einheit erscheint, kann es im Inneren durchaus sehr komplex strukturiert sein. In diesem Sinne bilden auch Objekte Hierarchien, die jedoch nicht systematischer Art sind.

Sehen und begreifen

Objekte werden aus verschiedenen Sichten betrachtet und dargestellt. Die Sicht auf ein Objekt wird dabei durch die Interessen und Ziele des Betrachters geprägt. Mit

zunehmendem Interesse an einem Objekt nimmt auch die Zahl der Sichten auf dieses Objekt zu. Eben die Art und Weise, die Sicht, aus der ein Sachverhalt begriffen wird, wird durch einen Begriff ausgedrückt, der die Art der Sicht, bezeichnet. In diesem Sinn implizieren Begriffe zur Bezeichnung von Arten immer die Sicht, mit der ein Sachverhalt gesehen wird. Diese Sicht wird durch das bestimmt, was in einem bestimmten Zusammenhang an diesem Sachverhalt besonders interessant ist. Es findet also durch die spezielle Sicht, durch den Begriff, dem der Sachverhalt zugeordnet wird, eine Einschränkung dessen statt, was der Sachverhalt tatsächlich ist. Die meisten Sachverhalte, insbesondere die Objekte, werden durch eine Vielzahl von Begriffen, die jeweils für eine Art oder Sicht stehen, beschrieben.



Sichten und Objekt

Eine bestimmte Sicht auf ein konkretes Objekt wird in seinem Zustand abgebildet. In diesem Sinn beschreibt eine Aussage über ein Objekt einen Objektzustand aus einer bestimmten Sicht. In der Sprachwelt werden also nicht Sachverhalte, sondern *Sichten auf Sachverhalte* bzw. Zustände bezüglich dieser Sichten abgebildet. Auf Objekte kann es zwar eine Vielzahl von Sichten und somit eine Vielzahl von Darstellungen geben. Das Objekt an sich existiert jedoch nur einmal. Während die verschiedenen Sichten auf ein Objekt durch entsprechende Begriffe zu Arten zusammengefaßt werden, gibt es keinen Begriff für *das Objekt an sich*. Objekte an sich werden nicht systematisiert. Sie werden nur durch verschiedene Sichten unterschiedlichen Systematiken zugeordnet. Begriffe zur Bezeichnung von Arten beziehen sich also auf Objektsichten, nie auf die Objekte selbst.

Vom Objekt zum Subjekt

Bestimmte Objekte können eine eigene Sicht auf die sie umgebende Welt entwickeln. Diese Objekte werden als Subjekte bezeichnet. Indem ein Subjekt eine eigene Sicht auf die Welt entwickelt, entscheidet es, welche Objekte aus welcher Sicht abgebildet werden. Die Art und Weise dieser Abbildungen wird in starkem Maße durch die Vorstellungen, Interessen und Wünsche des Subjektes geprägt. In diesem Sinn definiert das Subjekt seine eigene Weltsicht oder Weltanschauung.

Jede Welt, die in der Sprachwelt dargestellt wird, ist also in einem Subjekt und somit in einem RealObject begründet. Da ein Subjekt gleichzeitig Objekt ist, kann es in den eigenen oder in auch in den Augen anderer Subjekte wieder als Sachverhalt, als Gegenstand einer Darstellung erscheinen. Die Darstellung von Welten aus der Sicht eines Subjektes wird zunehmend eine Rolle spielen, da nur auf diesem Wege eine Kommunikation zwischen verschiedenen Subjekten möglich ist. Somit wird mehr und mehr das Bedürfnis entstehen, daß Subjekte in der Datenwelt Sichten über Sachverhalte austauschen können, wie es in der Sprache fast mit Selbstverständlichkeit geschieht.

2.4.1.2 Die Darstellung von Zusammenhängen

Die Sprache beschreibt nicht nur einzelne Sachverhalte, sondern bildet auch Zusammenhänge zwischen Sachverhalten ab. Ein Zusammenhang ist selbst wieder ein Sachverhalt, der eine Beziehung zwischen verschiedenen anderen Sachverhalten beschreibt. Zusammenhänge werden i.allg. zwischen konkreten Sachverhalten hergestellt. Sie selbst sind ebenfalls konkrete Sachverhalte. Genauso wie Objekte werden Zusammenhänge nach Arten klassifiziert, die die Sicht auf einen Zusammenhang definieren. Da der Zusammenhang an sich schon eine Systematisierung, also eine bestimmte Sicht darstellt, werden Zusammenhänge oft mit Sachverhalten einer Art assoziiert. Das OSM von ODABA2 kennt derzeit drei Kategorien von Zusammenhängen, die spezielle Zusammenhänge von Sachverhalten beschreiben.

- Sichten

  Sichten stellen einen Zusammenhang zwischen untergeordneten Sachverhalten dar. In einer Sicht werden eine Reihe von Merkmalen zusammengefaßt, die einen Sachverhalt detailliert darstellen. Jedes dieser Merkmale stellt wieder einen Sachverhalt dar, der einen Teil des übergeordneten Sachverhaltes beschreibt. Auch die Merkmale werden als Sachverhalte wieder aus einer bestimmten Sicht dargestellt, die ihrerseits wieder einen Zusammenhang untergeordneter Merkmale beschreiben kann.

- Mengen

  Mengenzusammenhänge werden durch Mengen gleich- oder verschiedenartiger Sachverhalte gebildet. In diesem Fall besteht der Zusammenhang einfach darin, daß die Sachverhalte einer gemeinsamen Menge angehören. Die Anzahl der Sachverhalte in einem Mengenzusammenhang ist dabei i.allg. nicht festgelegt. Mengen können selbst als Merkmale im Zusammenhang von Sichten, aber auch in anderen Zusammenhängen auftreten. Letztlich ist jede Menge in einem Zusammenhang definiert, der zumindest durch das Subjekt gegeben ist, das diese Darstellung der Sachverhalte in einer Menge begründet. Ein Mengenzusammenhang kann eine eigene Qualität besitzen und so mit zusätzlichen Merkmalen versehen sein. Das können z.B. Eigenschaften sein, die allen Sachverhalten der Menge eigen sind. In diesem Fall stellt der Zusammenhang eine Sicht dar, in deren Rahmen die entsprechenden Merkmale abgebildet werden können.

- Funktionale Zusammenhänge

  Auch funktionale Zusammenhänge sind Mengenzusammenhänge. Funktionale Zusammenhänge sind auf einer Menge von Sachverhalten (Argumenten) definiert. Dabei stellt jeder Sachverhalt eine Basissicht des funktionalen Zusammenhangs dar. In diesem Sinn ist ein funktionaler Zusammenhang von seinen Basissichten abgeleitet. Funktionale Zusammenhänge können eine eigene Qualität besitzen, die über Merkmale des Zusammenhangs dargestellt wird.

Über funktionale Zusammenhänge kann kooperatives Verhalten dargestellt werden. Die kooperierenden Sachverhalte werden in diesem Zusammenhang gemeinsam dargestellt und sind somit in der Lage, das entsprechende Verhalten auszuführen. Da die Zustände der beteiligten Sachverhalte bekannt sind, kann die entsprechende Auskunft oder Wirkung in diesem Zusammenhang bestimmt werden.

2.4.1.3 Die Ordnung der Arten

Das grundlegende Anliegen der Bildung von Arten besteht darin, Sachverhalte zu gruppieren, sie nach Arten zusammenzufassen. Darüber hinaus stellen die Arten selbst eine komplexe und vielseitige Systematik dar, die die Beziehungen zwischen Sachverhalten unterschiedlicher Arten beschreiben. Zum einen stehen die Arten in vielfältigen hierarchischen Beziehungen zueinander. Zum anderen werden über Arten Systematiken gebildet und Rollen von Sachverhalten dargestellt.

Die Ordnung der Arten basiert im wesentlichen auf der Bildung von Begriffshierarchien. Dabei gibt es drei unterschiedliche Verfahren, nach denen Arten gebildet werden können.

- Hierarchie der Arten

  Durch die Hierarchie der Arten werden Beziehungen kontextfreier Arten definiert. Kontextfreie Arten werden durch allgemeingültige Begriffe dargestellt.

- Spezialisierung zu Rollen und Merkmalen

  Durch die Spezialisierung zu Rollen oder Merkmalen werden kontextbezogene Arten gebildet, die bezüglich ihrer Basisart keine neuen Merkmale aufweisen.

- Bildung von Systematiken

  Über die Bildung von Systematiken werden Kategorien als abgeleitete Sichten zu einer definierten Basisart dargestellt.

Die Hierarchie der Arten

Arten bilden Hierarchien, die die Beziehungen zwischen verschiedenen Arten festlegen. Diese Beziehungen werden durch Ober- und Unterbegriffsbeziehungen oder sogenannte Art/Gattungsbeziehungen ausgedrückt. Diese Hierarchie entsteht dadurch, daß jeder Unterbegriff mit einer Reihe besonderer Eigenschaften oder besonderen Fähigkeiten verbunden ist – der Unterbegriff stellt eine Spezialisierung des Oberbegriffs dar. Somit können jedem Unterbegriff, jeder spezialisierten Art, weitere Merkmale zugeordnet werden, welche die Besonderheiten gegenüber dem allgemeineren Oberbegriff beschreiben. Durch die Hierarchie der Arten wird es also möglich, Sachverhalte in unterschiedlicher Genauigkeit darzustellen.

Sachverhalte in bestimmten Rollen

In konkreten Zusammenhängen können Sachverhalte eine bestimmte **Rolle** spielen. Durch die Rolle wird die Funktion oder Stellung bezeichnet, die der Sachverhalt in dem Zusammenhang hat. Im Zusammenhang mit einer PERSON kann eine andere PERSON z.B. die Rolle des **Vaters** oder der **Mutter** spielen.

Einer Rolle liegt immer eine Basisart zugrunde, die festlegt, wie der in der Rolle dargestellte Sachverhalt beschrieben werden soll (z.B. der Vater als Person). In diesem Sinne sind Rollen von ihrer Basisart abgeleitet (ein **Vater** ist eine PERSON). Diese Ableitungen sind jedoch insofern spezieller Art, als daß sie keine neuen Merkmale erhalten. Sie sind also genau durch die Merkmale ihrer Basisart beschrieben. Derartige Spezialisierungen einer Art werden auch **Rollenspezialisierung** genannt.

Rollenspezialisierungen zeichnen sich vor allem durch ein spezielles Verhalten des Sachverhaltes in seiner Rolle aus. Rollen werden durch kontextbezogene Begriffe, die nur in dem entsprechenden Zusammenhang eindeutig sind, bezeichnet. Rollenspezialisierungen können dazu führen, daß Merkmale der Art, die bisher variabel waren, in einer bestimmten Rolle hinsichtlich ihres Wertevorrats eingeschränkt oder sogar zu festen Merkmalen werden. So ist das **Geschlecht** ist zwar ein Merkmal von PERSONEN. Während es für beliebige Personen jedoch die

Werte *männlich* oder *weiblich* annehmen kann, liegt das Geschlecht für PERSONEN in der Rolle des **Vaters** oder der **Mutter** fest, wird also zum festen Merkmal.

Merkmale als Rollenspezialisierung einer Art

Im Zusammenhang mit einer Sicht wird die Rolle eines untergeordneten Sachverhaltes als Merkmal bezeichnet. **Merkmale** drücken spezielle Sichten auf untergeordnete Sachverhalte im Kontext einer Sicht aus. Merkmale sind von der Art, in der sie beschrieben werden, also von ihrer Rolle, abgeleitet. Wie Rollen besitzen auch Merkmale als abgeleitete Sichten keine zusätzlichen Merkmale. Die für Merkmale verwendeten Bezeichnungen (Begriffe) sind nur im Zusammenhang der übergeordneten Sicht definiert.

Die mit einer Sicht verbundenen Merkmale werden wie die Arten mit Begriffen belegt. Während die Begriffe für Arten kontextfrei sind, sind die Begriffe für Merkmale nur innerhalb einer Sicht definiert, d.h. Merkmale sind kontextbezogen. Die Sicht bestimmt also, in welcher Weise der Begriff, der ein Merkmal beschreibt, zu deuten ist.

Bildung von Systematiken

Eine wesentliche und bisher stark unterbewertete Artbeziehung sind Systematiken. Immer wenn die Sachverhalte bezüglich einer Sicht in Kategorien gegliedert werden sollen, wird eine Systematik gebildet, in deren Rahmen alle Sachverhalte einer definierten Art genau einer der Kategorien dieser Systematik zugeordnet werden können (z.B. Einkommensklassen für PERSONEN).

Systematiken werden bezüglich einer Art, der Basisart gebildet. Zum anderen können die Kategorien einer Systematik selbst mit Sichten verbunden sein, die von der Basisart abgeleitet sind. Kategorien müssen jedoch nicht zwangsläufig die Ableitungen einer Sicht darstellen. Wenn eine Kategorie mit der Basissicht direkt zusammenfällt, erscheint die Kategorie als Beschreibung der Rolle, die der Sachverhalt im gegebenen Zusammenhang spielt. Kategorien, die eine Rollenspezialisierung der Basisart sind, werden **elementare Kategorien** genannt.

Systematiken, die nur aus elementaren Kategorien bestehen, heißen **elementare Systematiken**.

2.4.1.4 Die Systematik der Merkmale

Merkmale können nach zwei Gesichtspunkten systematisiert werden. Der eine Gesichtspunkt bezieht sich auf die Rolle, die ein Merkmal im Rahmen einer Sicht spielen kann. Der zweite behandelt unterschiedliche Kategorien von Merkmalen.

Merkmale in verschiedenen Rollen

Die Beschreibung der Sachverhalte erfolgt aus einer bestimmten Sicht, d.h. im Rahmen einer Art, durch eine Reihe von Merkmalen. Dabei sind die Merkmale mit unterschiedlichen Rollen verbunden.

- **feste Merkmale**

  Jede Sicht ist durch eine Reihe fester Merkmale definiert, die alle Sachverhalte dieser Art auszeichnen. Diese festen Merkmale werden als bekannt vorausgesetzt und bedürfen in der Sprache keiner weiteren Erwähnung, wenn Einigkeit zu dem mit der Art verbundenen Begriff besteht - sie sind begriffsinhärent. Mit der Zuordnung zu einer Art werden einem Sachverhalt also schon einige feststehende Merkmale zugeordnet, die auf der Ebene der Art definiert sind. So sind z.B. die **Kantenzahl** oder die Zahl der **Ecken** eines Vierecks feste Merkmale. Da feste Merkmale in Abstraktionen zu variablen Merkmalen werden können (die Anzahl der Kanten und Ecken in einem POLYGON als Abstraktion eines VIERECKS ist z.B. variabel), bereitet die sachgerechte Darstellung fester Merkmale gegenwärtig noch einige Schwierigkeiten. Die gelegentlich empfohlene Darstellung fester Merkmale in „Metaklassen“ ist genauso problematisch, wie die redundante Darstellung in den Instanzen dieser Klasse.

- **variable Merkmale**

  Variable Merkmale sind Merkmale, die im Rahmen einer Sicht definiert, in ihrem Wert jedoch nicht festgelegt sind. Um einen konkreten Sachverhalt zu beschreiben, wird er direkt oder indirekt einer Art zugeordnet, die die Sicht definiert, aus der Sachverhalt dargestellt werden soll. Dann werden verschiedene variable Merkmale des Objektes bezüglich dieser Sicht beschrieben, indem ihnen Werte zugeordnet werden.

- **freie Merkmale**

  Darüber hinaus ist es gelegentlich erforderlich, Sachverhalte durch zusätzliche Merkmale zu beschreiben, deren Rolle der Sicht nicht bekannt ist. Dabei handelt es sich um Merkmale, die Besonderheiten eines Sachverhaltes darstellen. Diese werden oft unter der Rubrik „Sonstiges“ zusammengefaßt. Merkmale, die in dieser Weise als Merkmale unbekannter Art zur Beschreibung eines Sachverhaltes herangezogen werden, heißen freie Merkmale.

Die Systematik der Merkmale

Merkmale einer Sicht lassen sich verschiedenen Kategorien zuordnen, die durch die Systematik der Merkmale definiert werden. Dabei kann jedes Merkmal einer Sicht genau einer dieser Kategorien zugeordnet werden. Die Systematik der Merkmale im ODABA2-Sprachmodell geht dabei an einigen Stellen über die im ODM angestrebte Systematik hinaus. Während das ODM im wesentlichen zwei Kategorien kennt, die den Eigenschaften(Attributen) und den Beziehungen(Relationships) entsprechen, faßt das ODABA2-Sprachmodell auch Bestandteile und Basisarten(BaseStructures) als Merkmale auf.

- **Eigenschaften**

  Eigenschaften sind die Merkmale von Sachverhalten, die zuerst ins Auge fallen. Sie beschreiben weitgehend den Zustand eines Sachverhaltes. Eigenschaften sind mit dem Sachverhalt fest verbunden. Sie existieren nur im Zusammenhang mit dem Sachverhalt und sind für diesen immer definiert. Eigenschaften sind immer an ihren Träger (Sachverhalt) gebunden. Sie werden durch Beobachtung gewonnen und entsprechend ihrer Art abgebildet oder aus bekannten Eigenschaften abgeleitet. Eigenschaften einer Sicht können feste oder variable Merkmale sein.

- **Basisarten**

  Arten bilden Hierarchien, die die Beziehungen zwischen Ober- und Unterbegriffen darstellen. Die Hierarchie der Begriffe wird auf die Hierarchie der Arten abgebildet, indem Unterbegriffe als **abgeleitete Arten** oder Spezialisierungen dargestellt werden, während die Oberbegriffe **Basisarten** bezeichnen. Bei der Spezialisierung einer Art „erbt“ die abgeleitete Art alle Merkmale und das Verhalten der Basisart. Im Rahmen der Spezialisierung einer Art kann es jedoch auch zur Spezialisierung der Merkmale oder des Verhaltens kommen. Die Spezialisierung zu abgeleiteten Arten kann sich auf verschiedene Weise vollziehen:

  - Die Sicht wird um weitere feste Merkmale erweitert.
  - Die Sicht wird durch neue variable Merkmale erweitert.
  - Die Merkmale oder das Verhalten einer Sicht werden bezüglich ihrer Art spezialisiert.
  - Variable Merkmale werden in ihrem Wertevorrat eingeschränkt oder zu festen Merkmalen der Art.

  Einem Unterbegriff können durchaus mehrere Oberbegriffe zugrunde liegen. Jede abgeleitete Art ist also eine Ableitung von einer oder mehrerer Basisarten (Mehrfachvererbung).

  Es gibt zwei Gründe, die Basisart als Merkmal der Sicht und nicht über die Hierarchie der Arten darzustellen. Bei der Darstellung als Merkmal ist die Basisart kontextbezogen, wodurch eine Rollenspezialisierung im jeweiligen Zusammenhang möglich wird. Der zweite Grund ist, daß die Beschreibung

einer Sicht unvollkommen ist ohne die Beschreibung ihrer Basisarten. Praktisch erfolgt die Benennung von Basisarten nicht über die Definition der Hierarchie, sondern im Rahmen der Definition einer Sicht. Daraus leitet sich später die Hierarchie der Arten ab.

- **Bestandteile**

  Bestandteile einer Sicht sind Merkmale, die einzelne oder Mengen von abhängigen Sachverhalten darstellen. Es handelt sich bei Bestandteilen nicht unbedingt um Objekte, sondern mehr um Bestandteile der Sicht. Dabei können sich Bestandteile einer Sicht auch auf innere Objekte des Sachverhaltes beziehen. Bestandteile sind, wie Eigenschaften, an ihren Träger gebunden, d.h. sie existieren nur solange, wie die Sicht existiert, in der sie definiert sind. Im Gegensatz zu Eigenschaften müssen Bestandteile jedoch nicht unbedingt existieren.

  Indem ein Sachverhalt als Bestandteil einer Sicht dargestellt wird, wird eine spezielle Beziehung zwischen Sachverhalten (Objekten) definiert: eine „ist Teil von"- oder PART OF-Beziehung. Diese Art der Darstellung von Bestandteilen basiert eigentlich auf einer Objektbeziehung zu den Teilen. Da die Beziehung eines Ganzen zu seinen Teilen jedoch eine andere ist, als die Beziehung zwischen gleichberechtigten Objekten, scheint es sinnvoll, zwischen Bestandteilen und Objektbeziehungen zu differenzieren. Allerdings geht bei dieser Sicht der Objektcharakter der inneren Objekte etwas verloren. Es ist also ein Unterschied, ob innere Objekte als Bestandteile einer Sicht oder als Teile des Objektes dargestellt werden.

- **Objektbeziehungen**

  Genauso wie Bestandteile können auch Beziehungen zu anderen Objekten eine wesentliche Rolle bei der Darstellung von Sachverhalten spielen. Objektbeziehungen sind an sich spezielle Zusammenhänge. Als Merkmale beschreiben sie einen Zusammenhang zwischen zwei Objekten. Als Merkmal werden nur binäre Objektbeziehungen, also Beziehungen, die zwischen zwei Objekten bestehen, dargestellt. Objektbeziehungen sind in diesem Sinne symmetrisch, d.h. die Objekte „kennen" sich gegenseitig.

Über Objektbeziehungen werden keine zusätzlichen Merkmale der Beziehung abgebildet. Eine Objektbeziehung beschreibt also wirklich nur den Sachverhalt, daß sich zwei Objekte „kennen“. Alle darüber hinausgehenden Objektbeziehungen werden als funktionale Zusammenhänge oder als Objekte dargestellt.

Objektbeziehungen werden jeweils aus einer bestimmten Sicht dargestellt. Die Beziehungen werden also nicht zwischen den Objekten an sich, sondern zwischen Sichten auf Objekte abgebildet. Das ist zum einen dem Umstand geschuldet, daß Objekte nicht an sich, sondern nur über Rollen oder Sichten in Beziehung treten, Objektbeziehungen werden nur über Sichten sichtbar. Zum anderen gehen Objekte Beziehungen in einer bestimmten Rolle ein, die es nahelegt, diese Beziehungen über Sichten darzustellen, die diese Rolle zum Ausdruck bringen.

Objektbeziehungen werden zwischen unabhängigen Objekten hergestellt. Dennoch können über Objektbeziehungen auch Abhängigkeiten dargestellt werden, die auf die Sichten bzw. Rollen eines Objektes Auswirkungen haben. Wenn über eine Objektbeziehung existentielle Abhängigkeiten dargestellt werden, kann sich diese Abhängigkeit sowohl auf eine bestimmte Objektsicht als auch auf das Objekt an sich beziehen.

2.4.1.5 Eine Sache, die sich verhält

Bisher wurden Sachverhalte hauptsächlich als Sache aufgefaßt und dargestellt. Dadurch wird ein statisches Bild des Sachverhaltes gezeigt, das eigentlich nur für einen Moment Gültigkeit besitzt. Die Sprache ist jedoch in der Lage, ausgehend von einem bestimmten Zustand auszudrücken, wie sich der Sachverhalt verändern, entwickeln wird. Diese Veränderung bzw. Entwicklung besteht darin, daß der Sachverhalt selbst seinen Zustand verändert oder Veränderungen des Zustandes anderer Sachverhalte auslöst. Diese Tatsache wird als Verhalten eines Sachverhaltes beschrieben. Erst indem über das Verhalten abgebildet wird, wie sich ein Sachverhalt selbst und zu anderen Sachverhalten verhält, wird die Beschreibung eines Sachverhaltes vollständig.

Verhalten im Rahmen einer Sicht

Das Verhalten ist vor allem eine Fähigkeit von Objekten. Über die verschiedenen Sichten auf Objekte wird es als artspezifisches Verhalten dargestellt, das allen Objekten einer Art gemeinsam ist. In diesem Sinn unterliegt auch das Verhalten einer Systematisierung nach Arten. Dabei wird jeweils das Verhalten dargestellt, das aus der entsprechenden Sicht relevant ist und durch die in der Sicht abgebildeten Zustände beschrieben werden kann. Das Verhalten ist wie die Merkmale kontextbezogen. Für das Verhalten verwendete Begriffe sind also nur im Rahmen einer Sicht definiert. Im Kontext einer anderen Sicht können sie etwas ganz anderes ausdrücken.

Praktisch sind auch elementare Arten und Systematiken mit einem Verhalten ausgestattet. Das ODABA2-Sprachmodell betrachtet dieses Verhalten jedoch als vordefiniert, so daß es keiner weiteren Darstellung bedarf. Wie die Merkmale wird auch das Verhalten artspezifisch, also im Kontext einer Sicht dargestellt. In diesem Sinne wird auch das Verhalten für funktionale oder Mengenzusammenhänge im Rahmen einer Sicht beschrieben, indem diese als Sicht dargestellt werden.

Auskünfte als Ergebnis des Verhaltens

Durch das Verhalten können Auskünfte erteilt werden, ohne daß das Verhalten zustandsverändernd wirkt. In diesem Fall verändert das Verhalten weder den Zustand des eigenen noch den anderer Sachverhalte. Da auskunftserteilendes Verhalten im Ergebnis die Darstellung eines Sachverhaltes liefert, rückt dieses Verhalten in die Nähe der Merkmale. Genau besehen ist es von einem Merkmal nicht zu unterscheiden, da die Darstellung eines Merkmals im Rahmen einer Sicht genauso erscheint, wie eine durch ein Verhalten, eine Funktion abgeleitete Darstellung.

Somit können Merkmale auch durch entsprechendes Verhalten dargestellt werden. Ob ein Sachverhalt als Merkmal oder mittels eines Verhaltens dargestellt wird, ist uninteressant. In jedem Fall liefert es eine Aussage zu einem Sachverhalt einer bestimmten Art. Auskunftserteilendes Verhalten ist also wie die Merkmale mit

einer Art verbunden, die beschreibt, wie der Sachverhalt, der als Auskunft übermittelt werden soll, dargestellt wird.

Die Folgen des Verhaltens

Oft ist das Verhalten eines Sachverhaltes jedoch nicht nur informativ, sondern bewirkt Veränderungen des eigenen Zustandes oder löst Zustandsveränderungen an anderen Sachverhalten aus. Die Wirkung besteht also darin, daß Sachverhalte aufgrund des Verhaltens ihren Zustand ändern. Dabei ist es vorerst uninteressant, wodurch das Verhalten ausgelöst wurde. Die Beschreibung der Wirkung erfolgt abstrakt und nicht auf einen konkreten Sachverhalt bezogen. Die Wirkung beschreibt also, was passiert, in welcher Weise sich der Sachverhalt verändert, wenn er sich in dieser oder jener Weise verhält. Die Art der Veränderungen steht meistens im direkten Zusammenhang mit dem Ausgangszustand des Sachverhaltes.

Generell kann man sagen, daß ein Verhalten Veränderungen nur an dem Sachverhalt bewirkt, der sich verhält. Es kann jedoch direkt oder indirekt Zustandsänderungen anderer Sachverhalte zur Folge haben, indem es diese z.B. zu einem bestimmten Verhalten auffordert. In diesem Sinne betreffen die Wirkungen des Verhaltens nicht nur den ausführenden Sachverhalt, sondern auch andere Sachverhalte. Durch direkte Aufforderungen an andere Sachverhalte werden dabei beabsichtigte Wirkungen, Folgewirkungen erzielt. Neben diesen „beabsichtigten“ Wirkungen können verschiedene unbeabsichtigte Wirkungen, die Nebenwirkungen, eintreten. Indem durch das Verhalten ausgelöste Zustandsveränderungen Ereignisse auslösen, können diese zu mittelbaren Reaktionen anderer Sachverhalte führen. Diese werden jedoch besser als Kausalitäten dargestellt, da der auslösende Sachverhalt von diesen Nebenwirkungen nichts weiß.

Wie Verhalten funktioniert

Dem Verhalten liegt ein bestimmter Ausgangszustand zugrunde, der durch den Zustand des Sachverhaltes bestimmt ist, der sich verhält. Auf der Basis dieses Ausgangszustandes wird ein Zielzustand im Ergebnis des Verhaltens hergestellt, der sich in der Auskunft niederschlägt oder in der Zustandsveränderung des Sachverhaltes sichtbar wird. Sowohl die Auskunft als auch die

Zustandsveränderung des Sachverhaltes können als Funktion auf der Basis der Zustandsmenge des Sachverhaltes dargestellt werden. In diesem Sinn kann sowohl die Auskunft als auch die Wirkung beschrieben werden. Allerdings gestaltet sich das Verhalten nicht immer in so einfacher Weise. Zum einen können verschiedene verhaltensinhärente Bedingungen die Ausführung eines Verhaltens verhindern. Zum anderen wird das Verhalten oft durch zusätzliche Einflußgrößen bestimmt, die neben dem Ausgangszustand des Sachverhaltes Einfluß auf das Verhalten haben.

Einflüsse auf das Verhalten

Nicht immer kann das Verhalten einfach aus dem Ausgangszustand des Sachverhaltes, der sich verhält, abgeleitet werden. Oft gibt es eine Anzahl von Einflußgrößen, die das Verhalten maßgeblich beeinflussen. Somit ist das Verhalten zwar durch den Ausgangszustand des Sachverhaltes, darüber hinaus jedoch auch durch bestimmte Einflußgrößen, also Zustände anderer Sachverhalte, bestimmt. Der Einfluß dieser Sachverhalte wird durch ihren Zustand bestimmt (z.B. das hat Gefälle einer Strecke oder die Bodenbeschaffenheit Einfluß auf das Rollen einer Kugel). Bei der Beschreibung von Einflußgrößen wird also nicht nur der einflußausübende Sachverhalt, sondern auch dessen Wirkung auf das Verhalten beschrieben. Das Verhalten ist insofern schwer darzustellen, als daß oft nicht alle Einflußgrößen bekannt sind. Somit findet meistens eine idealisierte Darstellung des Verhaltens statt.

Vererbung und Spezialisierung von Verhalten

Mit der spezielleren Sicht auf einen Sachverhalt kann sich auch das Verhalten ändern. In vielen Fällen ist dies jedoch nicht der Fall. Dann kann das Verhalten an die speziellere Sicht vererbt werden. Die Spezialisierung des Verhaltens sich sowohl auf die Bedingungen, an die ein Verhalten geknüpft ist, als auch auf das Verhalten selbst beziehen.

Kooperatives Verhalten

Bisher wurde im wesentlichen **individuelles Verhalten** beschrieben, d.h. Verhalten, das von einem Sachverhalt ausgeführt wird. Praktisch begegnet uns

jedoch vielfach **kooperatives Verhalten**, das dadurch gekennzeichnet ist, daß mehrere Sachverhalte an der Ausführung des Verhaltens beteiligt sind (z.B. kann, wenn zwei KUGELN zusammenstoßen, die Wirkung nur auf der Basis des Zustands (**Größe**, **Masse**, **Geschwindigkeit**) beider KUGELN ermittelt werden). Abstrakt gesehen stellt jede Addition oder Multiplikation zweier Zahlen ein kooperatives Verhalten dar. Kooperatives Verhalten zweier Sachverhalte wird oft durch binäre Operatoren dargestellt. Das geht jedoch nur, solange genau zwei Sachverhalte in das kooperative Verhalten einbezogen werden. Kooperatives Verhalten beliebig vieler Sachverhalte kann über funktionale Zusammenhänge dargestellt werden. Dabei ist es manchmal nicht ganz einfach, zu entscheiden, ob bei einem Verhalten ein zweiter Sachverhalt eine Einflußgröße darstellt oder ob das Verhalten ein kooperatives Verhalten der beteiligten Sachverhalte ist. Da ein funktionaler Zusammenhang selbst ein Sachverhalt ist, wird auch das kooperative Verhalten als Verhalten eines Sachverhaltes dargestellt, nur daß es sich bei diesem Sachverhalt eben nicht um ein Objekt handelt.

2.4.2 Das Kausalitätsprinzip

Wir haben das Verhalten bisher als potentielle Fähigkeit betrachtet, die an sich noch nichts bewirkt. Das Verhalten bedarf in irgendeiner Weise eines Auslösers, der ein bestimmtes Verhalten aktiviert. Eine Möglichkeit, diesen Zusammenhang darzustellen, ist das Kausalitätsprinzip. Das Kausalitätsprinzip bildet Ursache/Wirkungs-Beziehungen zwischen Sachverhalten ab. Kausalitätsbeziehungen basieren auf der Auffassung, daß jeder Wirkung eine Ursache vorausgeht. In der Sprache werden kausale Beziehungen meistens als **wenn..., dann...** Konstrukte dargestellt. Dabei beschreibt das **wenn** die Ursache, einen bestimmten Zustand, das **dann** das ausgelöste Verhalten.

Zustandsübergänge für Sachverhalte

Während die Wirkungen durch entsprechendes Verhalten dargestellt werden, wird die Ursache als ein bestimmter Zustand oder Zustandsübergang aufgefaßt. Ein Zustand in der realen Welt entspricht einer bestimmten Situation zu einem bestimmten Zeitpunkt. Damit ist alles eingeschlossen, d.h. die Zustände aller Dinge

und ihre Beziehungen untereinander. Diese Zustände unterliegen einer ständigen Veränderung und sind in ihrer Komplexität nicht zu fassen. Reduziert man die Welt jedoch auf einen kleinen Ausschnitt, z.B. ein Bit, so sind die Zustände dieser Miniwelt, dieses Objektes, sehr wohl erkennbar und beschreibbar, nämlich {0,1} in diesem Fall.

Zustandsübergänge sind Zeitpunkte, in denen ein Sachverhalt bezüglich einer Sicht seinen Zustand ändert. An sich ist jeder Sachverhalt einer kontinuierlichen Zustandsveränderung unterworfen. Der größte Teil dieser Zustandsveränderungen ist jedoch uninteressant, wenn wir uns dem Objekt aus einer bestimmten Sicht nähern. Aus einer bestimmten Sicht heraus scheint der Sachverhalt bezüglich seiner relevanten Eigenschaften längere Zeit im gleichen Zustand zu verharren und nur zu bestimmten Zeitpunkten seinen Zustand zu verändern. Zustandsübergänge werden in diesem Sinne als Zeitpunkte aufgefaßt und nicht als Zeiträume oder Dauer. Der Widerspruch zwischen Wirklichkeit und Beobachtung, zwischen sprunghaften und kontinuierlichen Veränderungen resultiert vor allem daraus, daß wir uns bei der Beobachtung auf ausgewählte Merkmale beschränken und auch diese nur in einem gewissen Raster wahrnehmen. Dies ist zwar ein Mangel, der den gegenwärtigen Abbildungsmethoden anhaftet, doch so, wie man durch einige Punkte den Verlauf einer Kurve ziemlich genau andeuten kann, ist es auch möglich, durch Zustandsübergänge die Dynamik eines Modells recht gut zu beschreiben.

Ereignisse als relevante Zustandsübergänge

Allerdings ist nicht jeder Zustandsübergang eine Ursache. Die Zustandsübergänge, die Folgewirkungen nach sich ziehen und insofern Ursache sind, werden als Ereignisse bezeichnet. Ereignisse sind also Zustandsübergänge, die ein Verhalten, eine Aktion zur Folge haben können. Wirkungen sind dann Zustandsveränderungen, die infolge dieses Verhaltens eingetreten sind. In diesem Sinne wird durch das Kausalitätsprinzip beschrieben, unter welchen Umständen ein bestimmtes Verhalten eines Sachverhaltes aktiviert wird.

Kategorien für Kausalitätsbeziehungen

Nicht alle Kausalitätsbeziehungen lösen ein Verhalten aus. Genauso, wie bestimmte Ursachen ein Verhalten auslösen können, kann ein Verhalten auch verhindert werden. ODABA2 unterscheidet zwischen vier verschiedenen Kategorien von Kausalitätsbeziehungen:

- **Aufforderung**

  Wenn ein Sachverhalt sich in bestimmter Weise verhält, weil er durch einen anderen direkt dazu aufgefordert wurde, sprechen wir von Aufforderungen. Allerdings werden Aufforderungen derzeit meistens im Rahmen des Verhaltens und nicht als kausale Zusammenhänge dargestellt.

- **Reaktion**

  Bestimmte Zustandsübergänge (Ereignisse) von Sachverhalten führen in verschiedenen Fällen zu Reaktionen anderer Sachverhalte und lösen somit ebenfalls ein Verhalten (Aktion) aus.

- **Bedingung**

  In bestimmten Zusammenhängen sind Sachverhalte nur bedingt in der Lage, ein bestimmtes Verhalten auszuführen. Solche „äußeren Umstände“ werden als kausale Bedingungen bezeichnet. Bedingungen wirken nicht verhaltensauslösend, sondern verhaltensverhindernd.

- **Problem**

  Bestimmte Situationen (Zustände) führen zu Spannungen und drängen nach einer Lösung. Diese werden als Problem oder Problemsituation bezeichnet. Durch eine Folge von Zustandsübergängen beschreibt das Problem den Weg zur Lösung.

Es ist offensichtlich, daß nur die Aufforderung und das Ereignis vehaltensauslösend wirken. Bedingung und Problem wirken höchstens verhaltensverhindernd, indem eine bestimmte Bedingung das Ausführen eines Verhaltens verbietet oder eine nicht vorhandene Problemsituation ein Verhalten überflüssig macht.

2.4.2.1 Aufforderungen

Die einfachste Ursache ist eine direkte Aufforderung. In diesem Fall ist die Ursache kein Zustand oder Zustandsübergang, sondern ein Verhalten. Im Falle einer Aufforderung liegt die Ursache also nicht in einer Zustandsveränderung, sondern in dem Verhalten eines anderen Sachverhaltes. Indem ein bestimmter Sachverhalt durch das Verhalten eines anderen Sachverhaltes aufgefordert wird, sich in dieser oder jener Weise zu verhalten, kann sein Verhalten allerdings zu Zustandsänderungen führen, die ihrerseits Ursache weiterer Reaktionen sein können. Diese Zustandsänderungen können an dem aufgeforderten Sachverhalt selbst oder an anderen Sachverhalten erfolgen.

Aufforderungen richten sich direkt an einen oder mehrere Sachverhalte. Der auffordernde Sachverhalt kontrolliert dabei i.allg. das Ergebnis des angeforderten Verhaltens. Ein Sachverhalt, der einer Aufforderung Folge leistet, kann in seinem Verhalten wieder Aufforderungen an andere Sachverhalte richten, die dann ihrerseits mit einem entsprechenden Verhalten reagieren. Es kann sein, daß ein aufgeforderter Sachverhalt einer Aufforderung nicht nachkommen kann, weil er entweder nicht in der Lage ist, das Verhalten auszuführen (verhaltensinhärente Bedingung), oder durch die Umstände verhindert wird (kausale Bedingung).

Das durch die Aufforderung ausgelöste Verhalten ist zwar an verschiedene verhaltensinhärente und kausale Bedingungen gebunden und somit ebenfalls von den konkreten Umständen (Zuständen) abhängig, doch treten diese nicht als Bestandteile des kausalen Zusammenhangs der Aufforderung auf. Die Aufforderung wird durch den auffordernden und den aufgeforderten Sachverhalt sowie durch das angeforderte Verhalten beschrieben.

2.4.2.2 Reaktion auf Ereignisse

Das Verhalten wird nicht immer direkt durch Aufforderungen ausgelöst. In vielen Fällen verhalten sich Sachverhalte selbständig, indem sie auf bestimmte Ereignisse reagieren. Ereignisse sind etwas, was auffällt, unser Interesse erregt und eine Reaktion zur Folge hat. Ein Ereignis tritt ein, wenn ein Sachverhalt in bemerkenswerter Weise seinen Zustand verändert. Bemerkenswert bedeutet dabei,

daß die Zustandsveränderung „bemerkt“ wird. In diesem Sinne sind Ereignisse Ursache von Reaktionen.

Ereignisse können Reaktionen anderer Sachverhalte auslösen, ohne daß sie diese direkt auffordern. Ein Ereignis wirkt somit indirekt auf all die Sachverhalte, die dieses Ereignis registrieren und darauf reagieren. Dabei nimmt der auslösende Sachverhalt keine Kenntnis von der ausgelösten Reaktion. Im Gegensatz zu Aufforderungen, die andere Sachverhalte zu gezielten und kontrollierten Reaktionen veranlassen, können über Ereignisse Reaktionen ausgelöst werden, von denen der auslösende Sachverhalt nichts „weiß“. Dabei wirken Ereignisse meistens unmittelbar, d.h. die Reaktion tritt sofort ein.

Die Reaktion beschreibt den Zusammenhang zwischen einem Ereignis und dem ausgelösten Verhalten. Dabei ist das Ereignis als Zustandsübergang genauso einem Sachverhalt zugeordnet, wie das Verhalten, die Aktion. Ereignis und Aktion werden unabhängig für den auslösenden und die reagierenden Sachverhalte dargestellt. Erst die Reaktion beschreibt den (kausalen) Zusammenhang zwischen einem Ereignis und der ausgelösten Aktion.

Die an einer Reaktion beteiligten Sachverhalte, der ereignisauslösende und der reagierende Sachverhalt, können in unterschiedlicher Beziehung zueinander stehen. Ein Sachverhalt kann auf eigene Ereignisse, auf Ereignisse verwandter Sachverhalte, also Sachverhalte, zu denen es eine inhaltliche Beziehung gibt, und auf fremde Sachverhalte reagieren. In den ersten beiden Fällen sind die beobachteten Sachverhalte bekannt, so daß die Reaktion bezogen auf konkrete Sachverhalte erfolgt. Im letzten Fall reagiert der Sachverhalt auf eine bestimmte Art von Sachverhalten, er reagiert auf arttypische Ereignisse.

Der durch das Ereignis entstandene Zustand wirkt nur auslösend auf das Verhalten. Die Reaktion auf Ereignisse hat also nicht das Ziel, den eingetretenen Zustand zu verändern. Es ist bezüglich des Ereignisses weder festgelegt, welcher Art das ausgelöste Verhalten sein muß, noch ist bekannt, welche Sachverhalte auf ein Ereignis reagieren. Somit ist der ursächliche Sachverhalt auch kaum in der Lage, die eintretenden Reaktionen zu kontrollieren. Reaktionen werden als Zusammenhang zwischen möglichen Ereignisse und Verhalten auf der Schemaebene dargestellt. Tatsächlich treten Ereignisse aber auf der Sachebene ein.

Indem der Zusammenhang zwischen einem potentiellen Ereignis und dem mit diesem Ereignis verbundenen Verhalten definiert wird, kann das Verhalten beim Eintreten eines Ereignisses im voraus dargestellt werden.

2.4.2.3 Kausale Bedingungen

In einigen Zusammenhängen ist Verhalten an Bedingungen gebunden. Ein bestimmter Sachverhalt ist zwar potentiell in der Lage, ein Verhalten auszuführen, jedoch hindern ihn äußere Umstände. Eben diese Umstände sind nicht Bestandteil des arttypischen Verhaltens, sondern stellen einen kausalen Zusammenhang dar. Dieser kausale Zusammenhang wird durch Bedingungen beschrieben, die im jeweiligen Zusammenhang definieren, ob ein Verhalten ausgeführt werden kann oder nicht. Bedingungen sind Zustände, die eingetreten sein müssen, damit ein bestimmtes Verhalten ausgeführt werden kann. In diesem Sinn wirken Bedingungen nicht auslösend auf ein Verhalten, sondern verhaltensverhindernd. Die Ursache für das Auslösen eines Verhaltens (Aufforderung, Ereignis) steht in keinem unmittelbaren Zusammenhang mit der Bedingung.

Oft ist es eine Ermessensfrage, ob eine Bedingung kausal oder verhaltensinhärent ist. Während verhaltensinhärente Bedingungen die Voraussetzungen für die Möglichkeit, ein Verhalten auszuführen, definieren, legen Kausalitätsbedingungen die Umstände fest, unter denen ein Verhalten tatsächlich ausgeführt bzw. verhindert wird. In diesem Sinne beschreiben kausale Bedingungen kontextbezogene Voraussetzungen zur Ausführung eines Verhaltens, während verhaltensinhärente Bedingungen die grundsätzlichen Voraussetzungen für die Ausführung des Verhaltens festlegen.

Wie bei Ereignissen gibt es unterschiedliche Beziehungen zwischen dem Zusammenhang oder Sachverhalt, auf dessen Grundlage die Bedingung definiert ist - dem einschränkenden Sachverhalt - und dem Sachverhalt, der das Verhalten ausführt - dem reagierenden Sachverhalt. Zum einen können einschränkender und reagierender Sachverhalt identisch sein. In diesem Fall wird die Bedingung meistens als verhaltensinhärente Bedingung dargestellt. Es ist jedoch auch möglich, daß der einschränkende Sachverhalt mit dem reagierenden Sachverhalt verwandt

ist. Aber auch völlig fremde Sachverhalte, zu denen keine inhaltliche Beziehung besteht, können einschränkend wirken.

2.4.2.4 Probleme und Lösungen

Ein etwas schwächerer kausaler Zusammenhang wird durch das Problem definiert. Eine Problemsituation, ein Problemzustand ist ein Zustand, der nach Veränderung, nach einer Lösung drängt. Der Lösungszustand besteht zumindest in der Negation des Problemzustandes. Im Gegensatz zum Ereignis ist das Problem immer auf eine Lösung gerichtet, die den Problemzustand in einen Lösungszustand überführt. Die Lösung ist in diesem Sinn Bestandteil des Problems. Die hier dargestellten Probleme werden durch gezielten Zustandsübergänge, die durch einen oder eine Folge von Prozessen (Verhalten) herbeigeführt werden, dargestellt.

Zum Erreichen des Lösungszustandes muß die Lösung des Problems durch einen oder mehrere Prozesse herbeigeführt werden. In diesem Sinn erfordert auch die Lösung des Problems das Verhalten eines oder mehrerer Sachverhalte. Die Lösung eines Problems setzt oft mit einer zeitlichen Verzögerung zum Eintreten des Problemzustandes ein. Die Lösung eines Problems wird in Angriff genommen, wenn dazu aufgefordert wird oder wenn ein Ereignis eintritt, das die Lösung eines Problems unumgänglich macht. In diesem Sinn ähnelt die Lösung eines Problems einem komplexen Verhalten.

Probleme benötigen für ihre Lösung bestimmte Informationen, d.h. der Problemzustand sowie die Lösung des Problems basieren auf bestimmten Sachverhalten. Diese beschreiben das Problem insofern vollständig, als daß ein Problem in einem beliebigen Zusammenhang auftreten und gelöst werden kann, wenn diesem Zusammenhang die entsprechenden Sachverhalte, die Problemressourcen bekannt sind. Ein Problem kann also in verschiedenen Zusammenhängen auftreten und auf gleiche Weise gelöst werden. In diesem Sinn sind Probleme und ihre Lösungen portabel. Indem bei der Definition der Sachverhalte das Minimum dessen definiert wird, was zur Definition und Lösung des Problems erforderlich ist, kann ein hohes Maß an Portabilität für ein Problem erreicht werden.

Ein Problem kann in **Teilprobleme** zerlegt werden. Die Lösung eines Teilproblems stellt entweder den gewünschten Zielzustand teilweise her - **parallele Teilprobleme** - oder führt zu Zwischenzuständen, die erforderlich sind, um das nächste Teilproblem zu lösen - **sequentielle Teilprobleme**. Während im ersten Fall keine Reihenfolge für die Lösung der Teilprobleme vorgegeben ist, erfordert die schrittweise Lösung eines Problems, daß eine vorgegebene Folge der Teilschritte eingehalten wird. Eine Lösung kann auch in eine Menge **konkurrierender paralleler Teilprobleme** zerlegt werden, so daß die Lösung eines dieser Teilprobleme die Lösung des Problems ist. Auf diese Weise können verschiedene Lösungsstrategien dargestellt und gegeneinander abgewogen werden.

2.4.3 Das Temporalprinzip

Zustände eines Sachverhaltes sind immer auf einen Zeitpunkt bezogen. Tatsächlich ist es möglich, zwischen verschiedenen Zuständen zeitliche Zusammenhänge herzustellen. Es gibt ein Davor und ein Danach, es kann zu einem Sachverhalt eine zeitliche Reihe von Zuständen aus der gleichen Sicht dargestellt werden, die die Veränderungen dieses Sachverhaltes dokumentieren. Ein Sachverhalt befindet sich also nicht in einem bestimmten Zustand, sondern er befindet sich zu jedem Zeitpunkt in einem bestimmten Zustand. Die Sprache kann Darstellungen von Sachverhalten in einen zeitlichen Zusammenhang stellen. Das erweist sich z.B. dann als sinnvoll, wenn aus einer Folge von Zustandsveränderungen zukünftige Veränderungen prognostiziert werden sollen.

Die Darstellung temporaler Zusammenhänge ist also weitaus mehr, als die versionsbezogene Darstellung von Sachverhalten. Sie schließt das Erzeugen von Zeitereignissen (relevante Zustandsübergänge der Zeit) ebenso ein, wie zeitliche Zusammenhänge zwischen den Zuständen verschiedener Sachverhalte. Diese Zusammenhänge werden derzeit in ODABA2 nur in Ansätzen dargestellt.

Die Unterscheidung zwischen sehenden und gesehenen Sachverhalten (RealObjects und Instanzen) macht eine weitere Differenzierung erforderlich. Es ist tatsächlich ein Unterschied, ob sich der Sachverhalt selbst oder die Sicht auf den Sachverhalt ändert.

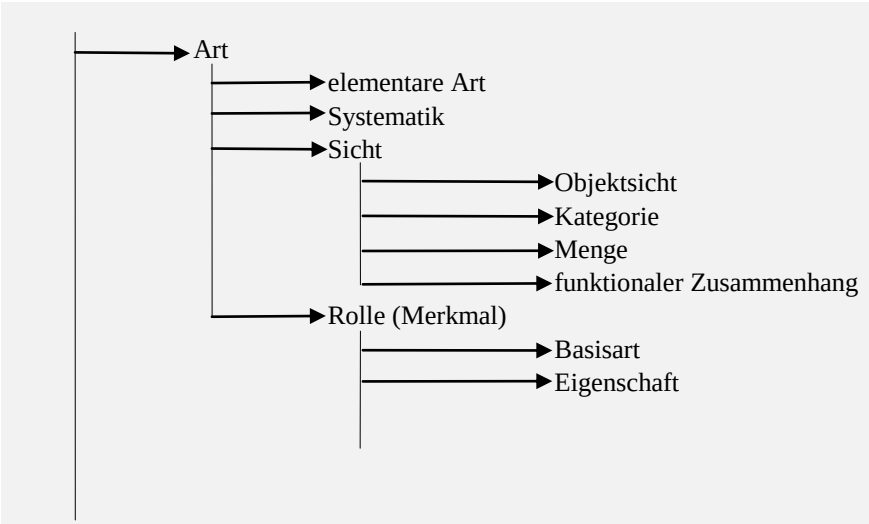
Beispiel Wenn *Anton* als Subjekt (RealObject) eine Sicht auf ein AUTO hat, die unter anderem den **Preis** des AUTOS einschließt, dann kann es sein, daß er der Annahme ist, das Auto koste DM 30 000. Zu einem späteren Zeitpunkt kann er feststellen, daß das Auto nur DM 28 900 kostet und seine Sicht präzisieren. Dann hat sich nicht der Zustand des Autos, sondern nur *Antons* Sicht darauf geändert, d.h. eigentlich hat sich *Antons* Zustand gewandelt.

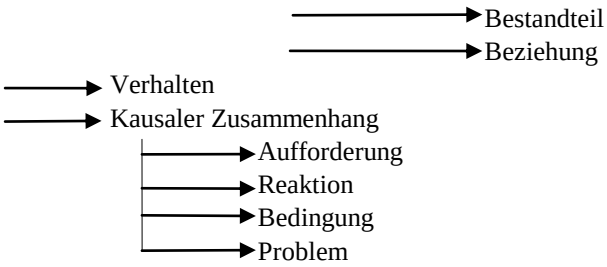
Eine solche Wandlung der Sicht liegt z.B. dann vor, wenn sich die Darstellung von Sachverhalten ändert. Auch in diesem Fall sind es nicht die Sachverhalte selbst, die ihren Zustand Ändern, sondern die Sicht, also die Art ihrer Darstellung.

Der Unterschied zwischen Versionen von Sachverhalten, die die Veränderung eines Sachverhaltes widerspiegeln, und der Veränderung von Sichten auf diesen Sachverhalt, die eine Veränderung des Subjekts, des Real Objects ausdrücken, wird durch Instanzen und Object-Versionen dargestellt.

2.4.4 Das objektorientierte Sprachmodell im Überblick

Die folgende Abbildung zeigt die Struktur des objektorientierten Sprachmodells. Es stellt die Arten auf der Modellebene und ihre hierarchischen Beziehungen dar. Die Pfeile sind in diesem Sinn als Begriffsableitungen zu verstehen (eine KATEGORIE ist eine ART). Sie zeigt auf diese Weise die Erweiterungs- und Entwicklungsmöglichkeiten des ODABA2-Sprachmodells auf.





Begriffshierarchie des objektorientierten Sprachmodells

Es ist offensichtlich, daß auch dieses Modell nur einen bestimmten Ausschnitt des Sprachmodells widerspiegelt und andere Systematiken durchaus denkbar sind. Die vorliegende Systematik des Sprachmodells wurde gewählt, um die Seite der Sprache sichtbar zu machen, die aufgrund ihrer Ähnlichkeit zum ODM besonders geeignet ist, um über den Weg der Sprachanalyse zum konkreten Objektmodell zu gelangen. Eine vollständige Modelldefinition dieses objektorientierten Modells wird im Anhang vorgestellt.

2.4.5 Sprachanalyse zur Bildung von Objektmodellen

Es gibt verschiedene Vorgehensweisen, um aus einem gegebenen Problem das technische Modell abzuleiten. Eine der bekanntesten ist das Prinzip der Object Modeling Technique (OMT). Dieses Herangehen ist zweckmäßig, wenn das Problem klar vor uns liegt. Wie aber gelangt man zur Klärung des Problems? Meistens ist dies ein umfangreicher Dialog zwischen dem Fach- und dem EDV-Spezialisten. Dieser Dialog wird im wesentlichen mit Mitteln der natürlichen Sprache geführt. Der Fachspezialist möchte sich an dieser Stelle nicht mit Konventionen der EDV-seitigen Darstellung beschäftigen. In dieser Phase ist also überwiegend die menschliche Sprache gefragt. Das Ergebnis wird in Form von Pflichtenheften oder Anforderungsspezifikationen niedergelegt. Dann erst beginnt die Phase des EDV-Analytikers, der diese Anforderungen systematisieren und mit den Mitteln der genannten Modelle darstellen soll. Erst jetzt werden die eigentlichen Probleme sichtbar, Mißverständnisse müssen geklärt werden, und eigentlich findet eine Fortsetzung der Problemanalyse statt.

ODABA2 bedient sich der Lexikonmethode, mit deren Hilfe ein Problem so analysiert werden kann, daß zum einen alle wichtigen Fragen beantwortet werden, zum anderen aus der Beschreibung direkt die technischen Modelle abgeleitet werden können. Dabei basiert die Lexikonmethode auf einem sprachlichen Ansatz. Obwohl dies etwas dem gegenwärtigen Trend nach graphischen Darstellungen widerspricht, bietet diese Methode weitreichende Vorteile.

- Da die Modellierung auf der Basis des allgemein bekannten Sprachmodells erfolgt, ist die Kommunikation auch mit dem Nicht-EDV-Spezialisten möglich.
- Die Lexikonmethode ist vollständig, d.h. sie ist in der Lage, alle Aspekte des Modellierungsprozesses darzustellen.
- Die Lexikonmethode ist beliebig erweiterbar und kann an verschiedene Modelle angepaßt werden.
- Mit der Lexikonmethode können auch umfangreiche und komplexe Zusammenhänge übersichtlich dargestellt werden.
- Aus der Darstellung können beliebige graphische Darstellungen abgeleitet werden.

Bei der Erarbeitung des Lexikons wurde großer Wert auf weitgehende Vollständigkeit und Abgeschlossenheit gelegt. Dadurch erscheint das Lexikon recht umfangreich. Es geht jedoch bei einer Schemabeschreibung nicht darum, alle Möglichkeiten des Modells zu nutzen, sondern die Sachverhalte den Stellen zuzuordnen, die ihnen entsprechen. Im einfachsten Fall läßt sich auch in diesem Modell eine Beschreibung auf der Basis des einfachen Prinzips der Arten realisieren, indem andere Möglichkeiten nicht genutzt werden. Insofern empfiehlt es sich, mit einer einfachen Darstellung zu beginnen, um dann schrittweise neue Möglichkeiten zu erschließen. Da jeder Schritt mit unmittelbaren Vorteilen bei der Implementierung der Anwendung verbunden ist, wird man sehr schnell die Grenzen dieser Methode erreichen und nach detaillierteren Möglichkeiten verlangen.

Das Lexikon einer Problembeschreibung besteht aus zwei Teilen, wenn wir das Temporalprinzip außer acht lassen:

- Beschreibung der Sachverhalte

  Die Beschreibung der Sachverhalte umfaßt sowohl die Sach- als auch die Verhaltensbeschreibung. Sie entspricht also etwa der Beschreibung des Objektmodells und des funktionalen Modells.

- Beschreibung der kausalen Zusammenhänge

  Hier werden die Zusammenhänge zwischen Ursachen und ihren Folgen beschrieben. Dieser Teil wird später im dynamischen Modell dargestellt.

Das funktionale Modell taucht nicht explizit auf, da es Bestandteil der Beschreibung der Sachverhalte ist. Tatsächlich sind die drei Modelle der OMT nicht ganz so streng voneinander zu trennen, da es bei der Beschreibung von Sachverhalten durchaus erforderlich sein kann, auf spezielle Ereignisse, die durch einen Sachverhalt ausgelöst werden, hinzuweisen oder darzustellen, unter welchen Umständen ein Sachverhalt in bestimmter Weise reagieren wird. Vielfach ist also die Beschreibung der Kausalitäten an die Beschreibung der Sachverhalte gebunden.

2.3.5.1 Das Lexikon

Lexika werden in ODABA zur Darstellung verschiedener Schemata verwendet, unter anderem auch zur Darstellung der Modellebene. Da Lexika ein wesentlicher Bestandteil der Dokumentation sind, wollen wir im folgenden etwas näher auf die Darstellung von Sachverhalten in einem Lexikon eingehen. Das Lexikon basiert auf einer zweistufigen Darstellung. Auf der ersten Stufe werden kontextfreie Begriffe wie ART oder REALOBJECT definiert. Im Kontext eines Begriffes werden dann wieder kontextbezogene Begriffe wie z.B. MERKMALE oder VERHALTEN einer Art definiert. Während die kontextfreien Begriffe eindeutig sein müssen, sind kontextbezogene Begriffe immer nur in ihrem Kontext bekannt und müssen auch nur dort eindeutig sein.

Jeweils links in der Zeile stehen die zu definierenden Begriffe. In dem folgenden Schemalexikon für eine PERSONEN-Darstellung werden ARTEN, **Merkmale** und ➲Schlüssel definiert.

PERSON[1]		Darstellung einer Person[2]
Name[3]	BEZEICHNER[4]	Vorname der Person
Familienname	BEZEICHNER	Familienname der Person
Alter	ZAHL	Alter in Jahren
Wohnsitz	ADRESSE	polizeilich gemeldeter Hauptwohnsitz
Personennummer	ZAHL	eindeutige Personennummer
➲IK_PERSON **Personennummer**		eindeutiger Schlüssel für Personen
➲SK_PERSON **Name** **Familienname**		Identifikation von Personen über Namen
ADRESSE		Darstellung einer Person
Straße	TEXT	Straßenname und Hausnummer
PLZ	ZAHL	Postleitzahl
Name	BEZEICHNER	Ortsname
Land	BEZEICHNER	offizielle Bezeichnung des Landes

Tabelle 3.2: Beispiel für ein Schemalexikon

Das Beispiel macht zwei Dinge deutlich. Zum einen ist zur Darstellung der Schemaelemente eine Symbolik erforderlich, über die eine Zuordnung zur entsprechenden Art des Modells hergestellt werden kann. In diesem Fall geht es um die Unterscheidung zwischen Arten, Merkmalen und Schlüsseln sowie um die Zuordnung zusätzlicher formaler Informationen. Es ist auch in einem normalen Lexikon üblich, Metainformationen über Symbole darzustellen. Da die Anzahl der Arten auf der Modellebene relativ klein ist, läßt sich diese Methode auf das

[1] An dieser Stelle steht in Kapitälchen die zu definierende Art.

[2] Erläuternder Text, der den Sachverhalt verbal beschreibt.

[3] Hier werden die Merkmale oder Kategorien beschrieben, die im Rahmen einer Sicht oder Systematik definiert werden sollen.

[4] Begriffe in Kapitälchen verweisen an dieser Stelle auf referenzierte Arten, die im Lexikon als Art definiert sein müssen.

Schemalexikon übertragen. In diesem Sinne sind die folgenden Symbole im einfachen artbezogenen Lexikon definiert:

ART	Bezeichnung einer Art. Die Kategorie der Art (Systematik. Sicht oder elementare Art) ergibt sich aus der jeweiligen Definition.
Merkmal	Bezeichnung eines Merkmals.
(n)	Das Merkmal kann n-fach auftreten (* – Anzahl undefiniert).
[ ]	Das Merkmal muß nicht unbedingt definiert sein.
➲	Definition eines Schlüssels.

Weitere Bezüge zum Modell sind in den Positionen der Angaben verschlüsselt (siehe Fußnoten).

Innerhalb eines Lexikons gelten bestimmte Regeln, die die formale Konsistenz und Vollständigkeit eines Lexikons festlegen. Für das artbezogene Sprachmodell lassen sich diese Regeln auf vier einfache Regeln reduzieren.

- Jede im Lexikon referenzierte Art muß im Lexikon definiert werden. Elementare Arten werden als im Modell bekannt angenommen. Die Bezeichnungen für Arten in einem Lexikon sind eindeutig.
- Jede Sicht ist durch mindestens ein Merkmal definiert. Die Bezeichnungen für Merkmale sind im Rahmen einer Sicht eindeutig.
- Jedes Merkmal einer Sicht ist mit einer Art verbunden.
- Jede Systematik ist durch mindestens eine Kategorie definiert.

Das Lexikon ist eine Methode der Sprachanalyse, mit deren Hilfe ein Problem sprachlich so beschrieben wird, daß daraus ein technisches Schema abgeleitet werden kann. Die formale Darstellung sowie die vier Konsistenzregeln sichern die formale Konsistenz der verbalen Problembeschreibung. Gleichzeitig wird die Verwendung gleicher Bezeichnungen für Arten und Merkmale in der Sprach- sowie in der Datenwelt unterstützt, was die Orientierung in der Datenwelt erheblich erleichtert. Auf diese Weise definiert das Lexikon ein **Natural Language Interface (NLI)**, über das die Kommunikation zwischen Sprach- und Datenwelt gesichert wird. Dies geschieht nicht nur, indem gleiche Begriffe benutzt und in

einer Dokumentation zusammengestellt werden, sondern indem auch im Rahmen einer Anwendung gesichert wird, daß auf diese Definitionen jederzeit zugegriffen werden kann. Das Lexikon hat also nicht nur eine Funktion in der Entwurfsphase, sondern stellt gleichzeitig auch die erforderlichen Informationen für den Anwender bereit, z.B. als Online-Hilfe oder in Form einer Dokumentation.

Der Vorteil eines Sprachmodells liegt darin, daß es weitgehend von technischen Details befreit ist. Es beschreibt das Problem in einer verständlichen Sprache und enthält doch schon alle wesentlichen Informationen, die zur Erzeugung eines Datenbankschemas erforderlich sind. Die Praxis zeigt immer wieder, daß nur das, was sich sprachlich klar ausdrücken läßt, in ein korrektes Datenbankschema überführt werden kann. Die sprachliche Klärung und Darlegung der Sachverhalte ist also nicht nur von dokumentarischem Wert, sondern wichtiger Bestandteil effizienter Analyse- und Darstellungsmethoden.

2.4.5.2 Beschreibung des Problems in einem Lexikon

KUGELKETTE		Eine Kugelkette besteht aus beliebig vielen durch Federn verbundenen Kugeln mit zwei Endpunkten, die durch die jeweils äußere Kugel gebildet werden. Durch die Federn werden die Kugeln jeweils in der Mitte zwischen ihren Nachbarn positioniert.
Kugeln(*)	KUGEL	Kugeln, aus denen die Kette besteht.
KUGEL		Eine beliebige Kugel, die als Kreis dargestellt wird.
Radius	ZAHL	Der Radius legt die Größe der Kugel fest.
Masse	ZAHL	Wir gehen vorerst von masselosen Kugeln aus. Bei der Verfeinerung des Schemas kann später auch die Masse der Kugel berücksichtigt werden.
Position	POSITION	Position der Kugel in der Ebene.
fixiert	ANZEIGE	Kugel ist durch eine äußere Kraft in ihrer Position fixiert. Die Endkugeln sind immer fixiert.
Nachbar-Nachbar[5]	KUGEL	Linker und rechter Nachbar der Kugel. Die Randkugeln haben nur einen linken bzw. rechten Nachbarn.
R **Balance** [6]	Positionswechsel	Wird bei einer Nachbarkugel ein Positionswechsel bemerkt, versucht die Kugel, die Balance wiederherzustellen.
E **Positionswechsel** [7]		Wenn die Kugel ihre Position verändert,

[5] Objektbeziehungen sind symmetrisch. Deshalb wird im Rahmen einer Beziehung immer der Rückbezug im Bezugsobjekt angegeben (z.B. Kinder-Eltern). Hier handelt es sich in beiden Fällen um das gleiche Merkmal.

[6] Im Rahmen der Objektbeziehung sollen alle Bezugsobjekte hinsichtlich des angegebenen Ereignisses (Positionsveränderung) beobachtet werden. Tritt das Ereignis ein, wird mit der entsprechenden Aktion (Balance) reagiert.

[7] Die Definition eines Ereignisses (E) definiert erst einmal nur das Ereignis, ohne Rücksicht darauf, wer das Ereignis später wahrnehmen wird.

			wird das als Ereignis signalisiert.
A **Balance** [8]	prüfen	v	Es muß geprüft werden, ob die Kugel beweglich ist. Wenn ja, kann sie verschoben werden.
	balancieren		Die Balance der Kugel wird wiederhergestellt.
P **verschieben** [9]	seriell		Bei der Positionsänderung einer Kugel stimmt die Zeichnung nicht mehr und muß aktualisiert werden.
L radieren [10]	KUGEL		Löschen der Abbildung in der alten Position.
L positionieren	KUGEL		Berechnen der neuen Position.
L zeichnen	KUGEL		Zeichnen der Kugel in der neuen Position.
V **prüfen**	ANZEIGER	a	Wenn die Kugel fixiert ist, kann sie nicht bewegt werden.
V **balancieren**		w	Die neue Position der Kugel wird aus den Positionen der Nachbarn berechnet. Anschließend wird die Kugel verschoben.
V **fixieren** [11]		w	Die Kugel kann nur noch durch äußere Kräfte bewegt werden. Sie schwingt nicht.
V **lösen**		w	Die Fixierung der Kugel wird gelöst.
V **zeichnen**		w	Die Kugel wird in ihrer neuen Position gezeichnet.
V **radieren**		w	Die Zeichnung der Kugel wird entfernt.

[8] Eine Aktion (A) wird durch die Vorbehandlung (v), das eigentliche Verhalten (ohne Markierung) und die Nachbehandlung (n) definiert, wobei in der Vorbehandlung kausale Bedinungen definiert werden können.

[9] Das P am Zeilenanfang verweist auf die Darstellung eines Problems. In diesem Fall wird das Problem durch serielle Teillösungen gelöst.

[10] Ein Teilproblem kann eine Lösung (L) sein. Dann besteht die Lösung in dem Verhalten eines entsprechenden Sachverhaltes. Dieses muß als Verhalten der angegebenen Art definiert sein.

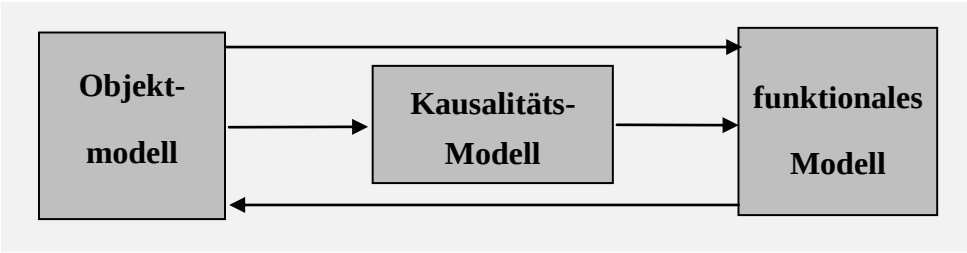
[11] Das V am Zeilenbeginn weist auf ein Verhalten hin, das im Rahmen der Sicht definiert wird. Das w vor der Beschreibung verweist auf die Beschreibung einer Wirkung. Ein zweiter Beschreibungsabschnitt könnte sich mit der Beschreibung der Auskunft befassen, wenn das Verhalten eine Auskunft liefert.

V **positionieren**		w	Die neue Position der Kugel wird durch Vektoraddition ermittelt.
POSITION			Position in der Ebene.
x	ZAHL		horizontale Position
y	ZAHL		vertikale Position
V **Mittelpunkt**	POSITION	a	Der Mittelpunkt zwischen zwei Punkten wird berechnet.
V +	POSITION	a	Die neue Position wird durch Addition eines Vektors berechnet.

2.5 Sprachwelt und Objektwelt

Ausgangspunkt der Entwicklung des Sprachmodells war es, Analogien zwischen dem Sprachmodell und dem Objektmodell herzuleiten, um aus sprachlichen Darstellungen technische Modelle ableiten zu können. In den folgenden Kapiteln wird gezeigt, wie das ODABA2-Sprachmodell technisch umgesetzt wird.

Bei der Spezifikation des ODM werden englische Bezeichnungen verwendet, um Unterschiede zwischen der Sprachwelt und der Objektwelt begrifflich zu fassen. Wie das objektorientierte Sprachmodell gezeigt hat, ist das ODM nicht nur ein strukturelles Modell, sondern wird in drei Teile gegliedert.



Teilmodelle des ODM

- Objektmodell

  Das Objektmodell definiert die verschiedenen Objektsichten und die strukturellen Zusammenhänge zwischen den Sichten. Es definiert die Regeln auf der Schemaebene zur Beschreibung der Sichten und Beziehungen der Sachverhalte zueinander, die als Strukturen und Strukturbeziehungen dargestellt werden.

- Kausalitätsmodell (Dynamisches Modell)

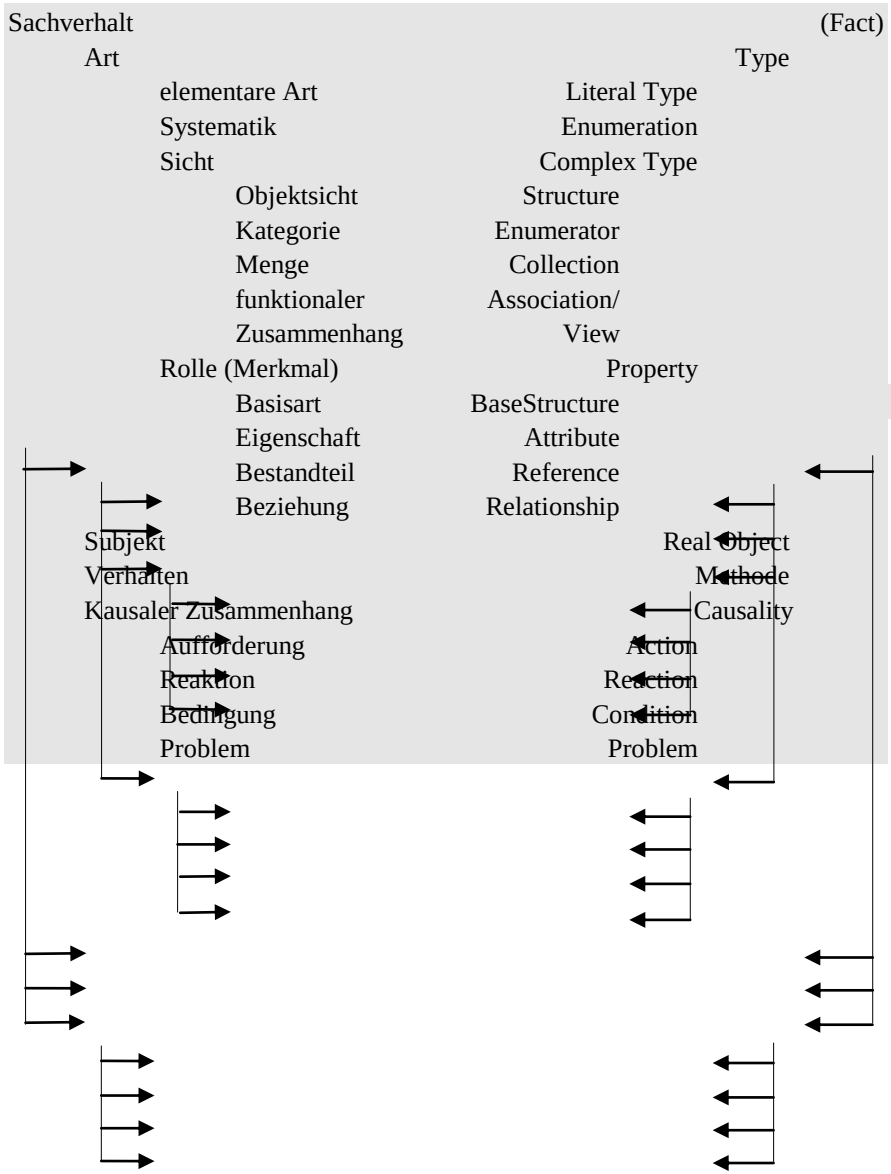
  Das Kausalitätsmodell oder auch das Dynamische Modell definiert die Regeln zur Darstellung von kausalen Zusammenhängen. Kausale Zusammenhänge beschreiben vor allem relevante Objektzustände bzw. Zustandsübergänge (Ereignisse) und damit verbundene Reaktionen.

- Funktionales Modell

  Das Funktionale Modell definiert die Regeln zur Beschreibung des Verhaltens der Sachverhalte. Neben Programmfunktionen gibt es vielfältige andere Möglichkeiten zur Darstellung von Verhalten.

Diese Gliederung berücksichtigt nicht das Temporalprinzip. Da die Darstellung zeitlicher Zusammenhänge nur in Ansätzen in ODABA2 realisiert wurde, werden sie hier nicht als Bestandteil des Modells behandelt.

Die folgende Gegenüberstellung zeigt die Analogie der Begriffe der ODABA2-Sprachwelt und der ODABA2-Objektwelt. Dabei sind die Begriffe der ODABA2-Objektwelt weitgehend dem ODMG-93-Standard angepaßt.


ODABA2-Sprachwelt
ODABA2-Objektwelt
Sachverhalt
(Fact)
Art
Type
elementare Art
Literal Type
Systematik
Enumeration
Sicht
Complex Type
Objektsicht
Structure
Kategorie
Enumerator
Menge
Collection
funktionaler Zusammenhang
Association/ View
Rolle (Merkmal)
Property
Basisart
BaseStructure
Eigenschaft
Attribute
Bestandteil
Reference
Beziehung
Relationship
Subjekt
Real Object
Verhalten
Methode
Kausaler Zusammenhang
Causality
Aufforderung
Action
Reaktion
Reaction
Bedingung
Condition
Problem
Problem

Kapitel 3
Das Objektmodell

In diesem Abschnitt wird der strukturelle Teil des ODABA2-Datenmodells (ODM) und seine Möglichkeiten beschrieben. Das Objektmodell schließt zum einen die Struktur der Daten, zum anderen ihre Darstellung in verschiedenen Zusammenhängen ein.

ODABA2 unterscheidet streng zwischen der Schema- und der Datenebene. Während auf der Schemaebene die Merkmale (Properties) der Instanzen definiert werden, bildet die Datenebene die Instanzen selbst ab. Instanzen stellen die konkreten Daten auf der Datenebene in verschiedenen Formen dar. Der Aufbau, die Struktur der Instanzen ist durch die Definition auf der Schemaebene bestimmt.

Die verschiedenen Abbildungsebenen können in ODABA2 durch verschiedene Datenbanken dargestellt werden. Genauso gut können Sach- und Schemaebene aber auch in einer Datenbank gespeichert werden.

Structure- und Property-Instanzen

Im ODM werden Daten in Instanzen zusammengefaßt. Eine Instanz kann atomar sein, also einen unteilbaren Wert enthalten, oder wiederum aus einer Menge von Instanzen bestehen. Type-Instanzen sind Instanzen, die zu einer Art gebildet werden. Structure-Instanzen sind spezielle Type-Instanzen, die für komplexe Types gebildet werden und aus einer Menge verschiedenartiger Property-Instanzen bestehen, die die Merkmale einer Sicht abbilden. Eine Property-Instanz kann ihrerseits aus einer oder mehreren Structure-Instanzen bestehen, die wiederum komplexer oder elementarer Art sein können. Dadurch ergibt sich eine alternierende Hierarchie von Structure- und Property-Instanzen, die zur Identifikation von Instanzen benutzt werden kann.

Während die Property-Instanzen einer Structure-Instanz über ihren Namen identifiziert werden (**Geburtsdatum**, **Name**, **Mitarbeiter**), werden Structure-Instanzen in Property-Instanzen durch Schlüsselwerte oder die Position der Instanz in einer Menge von Instanzen identifiziert. Diese Tatsache wird zur Definition einer allgemeinen Sprache, der Object Exchange Language (OEL) ausgenutzt, mit der sich jeder beliebige Sachverhalt als Instanzenpfad darstellen läßt.

Freie Instanzen

Instanzen, die physisch nicht an andere Instanzen gebunden sind, heißen freie Instanzen. Freie Instanzen können erzeugt, gelöscht und kopiert werden. Prinzipiell kann auf alle freien Instanzen ohne Einschränkung zugegriffen werden. In diesem Sinne stellen freie Instanzen eine Zugriffseinheit dar. Eingebettete Instanzen hingegen werden physisch in einer anderen Instanz, der Trägerinstanz, angelegt. Somit können sie nur im Zusammenhang mit ihrer Trägerinstanz bearbeitet werden.

Das Structure-Schema

Das Structure-Schema (ODABA2-Modell) ist hauptsächlich durch die Begriffe Structure und Property bestimmt. Entsprechend dem vorgestellten Sprachmodell gibt es jedoch noch einige weitere zentrale Begriffe.

- **Type**

  Types bilden die kontextfreien *Arten* des Sprachmodells ab. Dabei werden verschiedene Arten von Types als Literal, Enumeration oder Structure dargestellt.

- **Collection**

  Collections stellen Zusammenhänge als Mengen von unbenannten Instanzen dar. Dabei kann es sich um Instanzen gleicher oder unterschiedlicher Art handeln.

- **Property**

  Properties stellen *Merkmale* als benannte Instanzen im Kontext einer Structure dar. Im Zusammenhang mit Implementierungsklassen werden sie auch als Member bezeichnet.

- **RealObjects**

  RealObjects sind komplexe Zusammenhänge, die einen Sachverhalt aus verschiedenen Sichten, seine Sichten und seine Teile darstellen. Sie werden als benannte Instanzen abgelegt.

3.1 Types zur Darstellung von Arten

Types bilden die kontextfreien Arten des Sprachmodells ab. Ein Type ist also immer die Darstellung einer Art in der Objektwelt. Alle Instanzen in ODABA2-Datenbanken sind mit einem Type verbunden. Dabei kennt ODABA2 drei Kategorien von Types.

- **Literal**

  Literals stellen elementare Arten dar. Die Menge der darstellbaren elementaren Arten unterscheidet sich in den verschiedenen OODBS.

- **Enumeration**

  Enumerations stellen Systematiken dar. Dabei ist die Darstellung der Systematiken in den meisten Fällen auf elementare Systematiken beschränkt.

- **Komplexe Types**

  Komplexe Types sind Typen, deren Instanzen i.allg. nicht atomar, also aus mehreren Instanzen zusammengesetzt sind. Die wichtigsten komplexen Types sind Structures und Collections. Aber auch Associations und Views werden in ODABA als komplexe Types unterstützt.

Literals und Enumerations sind im Gegensatz zu komplexen Types atomare Types, deren Instanzen semantisch unteilbar sind.

3.1.1 Literal Types für elementare Arten

Eine Art atomarer Types sind Literal Types, durch die die elementaren Arten abgebildet werden können. Sie sind in dem Sinne elementar, als daß sie strukturell als untrennbare Einheit aufgefaßt werden. Dennoch verfügen sie oft über eine innere Struktur sowie auch über eine Basisfunktionalität, die meistens in Form von Operatoren bereitgestellt wird.

- **Atomic Literal**

  Atomic Literals sind untrennbare Types wie z.B. ganze Zahlen oder Zeichen.

- **Structured Literal**

  Structured Literals stellen zwar nach außen eine untrennbare Einheit dar, besitzen aber eine innere Struktur.

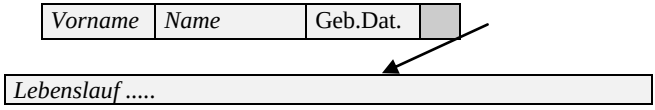
ODABA2 kennt sowohl Atomic Literals (INT, CHAR usw.) als auch Structured Literals (DATE, TIME). Der Zeitpunkt (Date_Time) wind in ODABA nicht als Literal Type sondern als Klasse bereitgestellt. Die Menge der Literal Types ist im Lexikon unter ➔**DataTypes** beschrieben.

Instanzen für Literals werden in ODABA2 nicht als freie Instanzen abgelegt, sondern als benannte Instanzen (Properties), also z.B. im Kontext einer Structure, erzeugt und direkt in der Structure-Instanz abgelegt. Eine spezielle Form des Literals sind lange Textfelder (MEMO). Dies sind in dem Sinne unstrukturierte Felder, als daß ihre Struktur nur durch ihren Inhalt bestimmt wird, der aus Text besteht. Aufgrund der stark variierenden Länge dieser Felder werden diese nicht direkt in der Structure-Instanz abgelegt, sondern als Referenz gespeichert.

Beispiel Wenn im Kontext einer einfachen Personenstruktur die Properties **Name**, **Vorname**, **Geburtsdatum** und **Lebenslauf** abgelegt werden sollen, wäre folgende Strukturdefinition denkbar:

```
structure Person {
   attribute (     Name       { type STRING; size 30 }
                   Vorname    { type STRING; size 30 }
                   Geburtsdatum { type DATE }
   )
   reference       Lebenslauf { type MEMO; size 2000 }
}
```

Daraus würde folgende Instanzenstruktur resultieren.



3.1.2 Enumerations für Systematiken

Systematiken werden im ODABA2 als Enumerations dargestellt. Über elementare Kategorien können Sachverhalte spezialisierten Arten zugeordnet werden, ohne daß zusätzliche Merkmale gebildet werden (Klassifikation). Durch elementare Kategorien werden also Rollen einer Sicht bezeichnet. Diese können in einer Enumeration als Enumerators dargestellt werden. Enumerations sind spezielle Types (➔**SDB_Type**).

Enumerators bestehen aus einem Bezeichner, der die Kategorie benennt, und einer internen Nummer, die in der Instanz als Wert gespeichert wird. Dabei kann bezüglich einer Systematik einer Property-Instanz genau ein Enumerator der Enumeration zugeordnet werden. In diesem Sinne ist gesichert, daß die gebildeten Kategorien bezüglich einer Menge von Sachverhalten disjunkte Teilmengen bilden.

Beispiel Im Zusammenhang mit der Darstellung von PERSONEN sind für den ABSCHLUß die Kategorien **Abiturient**, **Fachschulabsolvent**, **Hochschulabsolvent** und **Sonstige** (ist wegen der Vollständigkeit erforderlich) denkbar. Die Eindeutigkeit ist gegeben, wenn man jeweils den höchsten ABSCHLUß als Eigenschaft betrachtet. Diese Kategorien können in einer Domäne zusammengefaßt und als Enumeration definiert werden:

```
Enumeration Abschluss {
   enumerator ( Abiturient            { code 1 }
                Fachschulabsolvent    { code 2 }
                Hochschulabsolvent    { code 3 }
                Sonstige              { code 0 }   )
}
```

Da Enumerations auf der Schemaebene definiert werden, stellen sie innerhalb einer Anwendung eine unveränderliche Größe dar. Eine Erweiterung oder Modifikation des Wertevorrats bedeutet somit immer eine Schemaänderung.

Instanzen für Enumerations werden nur im Zusammenhang mit Properties, also im Kontext einer Structure, erzeugt und direkt in der Instanz der Structure abgelegt. Dabei wird die interne Nummer verwendet. Das führt nicht nur zur Einsparung an Speicherplatz, sondern auch zu einer effizienteren Verarbeitung, da sich numerische Werte einfacher verarbeiten lassen als Zeichenketten. Bei der Verwendung von Enumerations sichert ODABA2 durch die Tatsache, daß in dem

Feld nur ein Wert abgelegt werden kann, die Forderung nach der Bildung disjunkter Teilmengen bezüglich der Kategorien. Da für Systematiken gleichzeitig die Vollständigkeit der Kategorien gefordert ist, wird von ODABA2 gesichert, daß nur Instanzen gespeichert werden, die bezüglich ihrer Systematiken gültige Werte aufweisen. Jede Instanz muß also bezüglich eines Enumeration-Feldes mit einem gültigen Enumerator belegt sein. Das kann im Zweifelsfall durch einen Wert für eine Standardkategorie (z.B. **Sonstige**) erreicht werden.

Beispiel Wenn wir PERSONEN nach ihrem **Abschluß** systematisieren wollen, können wir eine einfache Personenstruktur mit den Properties **Name**, **Vorname** und **Abschluß** definieren.

```
structure Person {
   attribute (     Name       { type STRING; size 30 }
                   Vorname    { type STRING; size 30 }
                   Abschluss { type Abschluss }       )
}
```

Daraus würde folgende Instanzenstruktur resultieren:

Vorname	*Name*	3	2

Entsprechend der Definition der Enumeration ABSCHLUß bedeutet Hochschulabschluß.

Extended Enumerations

Die Möglichkeiten der Enumerations genügen in dieser Weise kaum den Anforderungen, die sich aus der Darstellung von Systematiken ergeben. Zum einen fehlt der Zusammenhang zwischen der Zuordnung zu einer Kategorie und der damit verbundenen Spezialisierung der Sicht. Deshalb können Enumerators in ODABA2 mit einem Type assoziiert werden.

Beispiel Wenn für ABITURIENTEN und FACHSCHULABSOLVENTEN die **Abschlußnote** und die **Abiturart** von Interesse sind, während für HOCHSCHULABSOLVENTEN sind die ABSCHLÜSSE in verschiedenen Fächern dargestellt werden sollen, kann dieser Sachverhalt durch folgende Definition dargestellt werden:

```
Enumeration Abschluss {
   baseStruct   person type Person;
   enumerator (
      Abiturient  { type Abiturient; code 1 }
      Fachschulabsolvent{ type FAbsolvent; code 2 }
      Hochschulabsolvent{ type HAbsolvent; code 3 }
```

```
        sonstige   { code 0 }    )
}
```

In diesem Fall müssen alle mit den Enumerators assoziierten Types direkt oder indirekt von der BaseStructure der Enumeration abgeleitet sein. Außerdem führt der Wechsel einer Kategorie in einer Instanz dazu, daß die bisherige abgeleitete Instanz gelöscht und eine andere abgeleitete Instanz entsprechend der neuen Kategorie erzeugt wird.

Extended Enumerations stellen somit bei ihrer Verwendung in einer Structure-Instanz Metainformationen auf der Instanzenebene bereit. In ODABA2 kann ein extended Enumeration-Attribute mit einer Reference oder Relationship der gleichen Structure verbunden werden. Auf diese Weise kann über das Enumeration-Attribute der Type der Reference festgelegt werden (siehe auch „3.2.1 Types und References“)

Abgeleitete Enumerations

Enumerations können in dem Sinne spezialisiert werden, als daß ihr Wertevorrat, die Menge der Enumerators, eingeschränkt wird. Das ist immer dann der Fall, wenn in bestimmten Zusammenhängen eine oder mehrere Kategorien nicht auftreten können. Oft besteht die Spezialisierung von Sichten gerade in der Einschränkung bestimmter Wertevorräte.

Beispiel Wenn PERSONEN nach ihren ALTERSGRUPPEN systematisiert werden (**Kinder**, **Erwachsene** und **Rentner**), kommen für HOCHSCHULABSOLVENTEN nur die Kategorien **Erwachsene** und **Rentner** in Frage. Die Spezialisierung zu HOCHSCHULABSOLVENT führt also zu einer eingeschränkten Enumeration für ALTERSGRUPPEN. Wir können jetzt also auf der Basis von ALTERSGRUPPEN eine abgeleitete Enumeration HALTERSGRUPPEN für HOCHSCHULABSOLVENTEN bilden:

```
Enumeration Altersgruppen {
   enumerator(  Kinder          { code 1 }
                Erwachsene      { code 2 }
                Rentner         { code 0 }      )
}
Enumeration HAltersgruppen {
   baseEnum     AG type Altersgruppen;
   enumerator ( Erwachsene; Rentner )
}
```

Die Verwendung von abgeleiteten Enumerations ist vor allem in Verbindung mit der Definition virtueller Properties sinnvoll. Eine detaillierte Darstellung der Definition von Enumerations ist im Lexikon unter zu finden.

3.1.3. Komplexe Types

Komplexe Types sind Types, deren Instanzen aus mehreren untergeordneten Instanzen bestehen. ODABA2 kennt folgende komplexe Types:

- **Structure**

  Structures bilden die verschiedenen Sichten des Sprachmodells ab. Durch Structures werden sowohl Objektsichten als auch funktionale Zusammenhänge (Association) dargestellt. Die Sichten werden in Structure-Instanzen abgebildet, die ihrerseits wieder eine Menge von benannten Property-Instanzen darstellen.

- **Association/View**

  Views werden zur Darstellung abgeleiteter Zusammenhänge gebildet. Da sie in diesem Sinne eine Methode zur Darstellung abgeleiteter Daten definieren, könnten sie ebenso als Methode aufgefaßt werden. Da ODABA2 aber persistente Views unterstützt, werden sie hier als komplexe Types aufgefaßt.

Komplexe Types sind als solche nicht implementiert, sondern treten nur in ihren Spezialisierungen auf.

3.1.3.1 Structures zur Darstellung von Sichten

Structures werden zur Darstellung von Sichten in der Objektwelt definiert. Durch Structure-Instanzen werden sowohl Sichten auf Objekte und Kategorien, als auch Sichten auf Zusammenhänge dargestellt. So, wie eine Sicht durch ihre Merkmale beschrieben wird, wird eine Structure durch ihre Properties dargestellt. Eine Structure stellt sowohl einen Type (**➔SDB_Type**) als auch kausalen Zusammenhang (**➔SDB_Causality**) dar. Kausale Zusammenhänge werden später im Zustandsmodell beschrieben.

Identifikation von Structure-Instanzen

Während die Property-Instanzen durch ihren Namen innerhalb einer Structure-Instanz identifiziert werden können, gestaltet sich die Identifikation von Structure-Instanzen etwas komplizierter, da Structure-Instanzen oft als unbenannte Instanzen in Mengen gespeichert werden.

Instanzen-Identity zur Identifikation von Instanzen

Die Instanzen-Identity ist eine eindeutige Bezeichnung, die eine Instanz in einer ODABA2-Datenbank oder auch darüber hinaus identifiziert. Alle freien Instanzen sind auf Lebzeiten mit einer Identity verbunden. Andrerseits wird eine einmal vergebene Identity auch nach dem Löschen einer Instanz nicht erneut vergeben. Instanzen-Identities werden nicht nur für Structure-Instanzen, sondern auch für Collection-Instanzen erzeugt. Da ODABA2 zuläßt, daß Objekte durch mehrere Sichten und somit durch verschiedene Structure-Instanzen dargestellt werden können, wird hier zwischen der bekannten Object Identity und der Identity von Instanzen unterscheiden. Die Instanzen-Identity kann zur Adressierung der Instanzen innerhalb der Datenbank verwendet werden, unabhängig davon, in welcher Collection eine Instanz referenziert wird.

Keys zur Identifikation von Structure-Instanzen

Zur Identifikation von Structure-Instanzen ist die Instanzen-Identity nur bedingt geeignet, da sie dem Anwender i.allg. verborgen bleibt. Außerdem ist vom Anwender, der die Sachverhalte auch sprachlich unterscheiden muß, innerhalb einer Menge von Structure-Instanzen meistens schon ein Identifikator zur eindeutigen Bezugnahme auf die Instanzen gebildet worden. Derartige Identifikatoren werden als Schlüssel bezeichnet und im Rahmen einer Structure als Key dargestellt. Keys müssen nicht immer eindeutig sein, sondern können zur Bildung von Teilmengen verwendet werden (klassifizierende Schlüssel). ODABA2 unterstützt die Definition eines identifizierenden Schlüssels, eines Ident Keys, über den Structure-Instanzen in beliebigen Mengen identifiziert werden können.

Schlüssel können durch ein oder mehrere transiente oder persistente Properties gebildet werden. Da transiente Properties abgeleitete Werte der Instanz enthalten können, die durch entsprechende Funktionen berechnet werden, können in ODABA2 auch Schlüssel über abgeleitete Werte gebildet werden. Die Anzahl der Schlüssel, die für eine Structure definiert werden können, ist praktisch nicht begrenzt.

Schlüssel können später verwendet werden, um Indizes über verschiedene Collections zu bilden. Die Definition eines Schlüssels ist jedoch ein rein logischer Vorgang und bedeutet noch nicht, daß für diesen Schlüssel Indizes erzeugt werden. Genauere Angaben zur Schlüsseldefinition können im Lexikon unter **➔SDB_Key** nachgelesen werden.

- **Indirekte Keys**

  Ein Key kann aus mehreren Bestandteilen zusammengesetzt sein. Dabei können die Bestandteile jeweils durch eine Property der Structure oder durch Properties referenzierter Structures dargestellt werden. Dadurch ist es möglich, auch referenzierte Structures in die Schlüsselbildung einzubeziehen (indirekte Keys). Indirekte Keys sind Keys, für die die schlüsselbildende Instanz nicht identisch ist mit der Instanz, für die der Key gebildet wird (z.B. der **Name** der **Mutter** als Key für PERSON).

- **Multiple Keys**

  Multiple Keys sind Keys, für die mehrere Schlüsselwerte zu einer Instanz erzeugt werden können, z.B. um ein Stichwortverzeichnis für Zeitschriften (Instanzen) zu bilden. Multiple Keys werden definiert, indem Mengen als Key Components angegeben werden. Dabei kann eine Menge durch einen Property Array oder eine Reference Collection dargestellt werden.

Versionsbildung für Structures

ODABA2 unterstützt die Bildung von Structure-Versionen. Dadurch wird es möglich, Instanzen zu verschiedenen Strukturständen in einer Datenbank zu speichern. Damit entfällt die Notwendigkeit einer Reorganisation bei Änderungen der Structure-Definition. Sind eine oder mehrere Strukturänderungen vollzogen worden, werden Structure-Instanzen älterer Versionen entsprechend dieser

Entwicklung konvertiert, bis sie dem aktuellen Stand entsprechen. Beim nächsten Speichervorgang werden sie dann entsprechend der aktuellen Structure-Version abgelegt.

Bei häufigen Srukturänderungen führt dieses Verfahren jedoch zu einer erheblichen Laufzeitbelastung, so daß in diesem Fall eine Reorganisation durchgeführt werden sollte.

Persistent Structure

Nicht alle Structures werden als Instanzen in der Datenbank dargestellt. Sollen jedoch Instanzen zu einer Structure in der Datenbank gespeichert werden, muß die Structure zur Persistent Structure (➔**SDB_ODABA_Str**) spezialisiert werden.

Persistente Strukturinstanzen sind Strukturinstanzen, die einen Prozeß überdauern, indem sie in der Datenbank abgelegt werden. Eine Structure kann zur persistenten Structure (ODABA-Struktur) spezialisiert werden. Instanzen persistenter Strukturen werden in der Datenbank gespeichert. Für persistente Strukturen gibt es spezielle Konstruktoren und Zugriffsverfahren, um auf ihre Strukturinstanzen zugreifen zu können. Außerdem ist für persistente Strukturen eine Ereigniskontrolle (**Event Control**) aktiv, die definierte Ereignisse bemerkt und signalisiert.

Generic Types oder Template Structures

Generic Types sind Structure-Definitionen, deren Properties mit variablen Types definiert werden können. Diese werden später durch aktuelle Typangaben ersetzt. Generische Typen werden vor allem für transiente Strukturen benutzt, um eine Menge von Instanzen zu einem Typ in geeigneter Weise zu organisieren (z.B. als Menge, sortierte Liste). ODABA2 unterstützt auch auf der Datenbankebene generische Typen. Ein Standardtyp, der bereitgestellt wird, ist INTERVAL.

Generische Typen werden als Type referenziert, indem sie mit den aktuellen Typvariablen parametriert werden (z.B. INTERVAL(REAL) - Intervall für Gleitkommawerte). Gegenwärtig unterstützt ODABA2 nur einen Parameter für Structure Templates.

3.1.3.2 Associations und Views für funktionale Zusammenhänge

Funktionale Zusammenhänge können je nach Komplexität als Associations oder Views dargestellt werden. Während Associations einen unmittelbaren funktionalen Zusammenhang zwischen verschiedenen Structures darstellen, können Views mit sehr komplexen Bildungsvorschriften definiert werden.

Associations

Ein Spezialfall funktionaler Zusammenhänge sin Relationships, die einen Zusammenhang zwischen zwei Sachverhalten, also einen binären Zusammenhang darstellen, in dem keine weiteren Merkmale abgebildet werden können. Ein allgemeineres Konzept ist die Darstellung von funktionalen Zusammenhängen als Associations. Associations erscheinen als eigenständige Definitionsobjekte und können sowohl Zusammenhänge beliebiger Ordnung beschreiben als auch aus dem Zusammenhang abgeleitete Merkmale als Properties darstellen.

Beispiel Der funktionale Zusammenhang zwischen einem PRODUKT und den eingehenden Halbprodukten oder Materialien (Produktmatrix) beschreibt die **Menge** des eingehenden PRODUKTES in das hergestellte PRODUKT.

```
structure ProduktMatrix {
   baseStruct (
      Material { type Produkt; based_on Produkt }
      Produkt  { type Produkt; based_on Produkt )
   attribute    Menge       type INT;
}
```

Material und Produkt sind in diesem Zusammenhang freie Structure-Instanzen, die nicht als eingebetteter Bestandteil der Structure-Instanz der Produktmatrix dargestellt werden.

Die Darstellung von Associations erfolgt über Structures und ihre Instanzen. Die Argumente des funktionalen Zusammenhangs werden dabei als Base Structures abgebildet. Die Properties der Association werden als Properties der Structure dargestellt. Dabei können nicht nur atomare Werte als Funktionswerte entstehen. Es ist ebenso möglich, daß Collections oder Structure-Instanzen als Werte eines funktionalen Zusammenhangs abgebildet werden.

Kontrollierte Domänen

Die Domäne einer Association, ihr Wertevorrat, ergibt sich theoretisch aus der Produktmenge der Basismengen für die Argumente. Dabei sind die Basismengen durch die Basis-Collections der einzelnen Base Structures definiert. Eine Association-Instanz ist also nur solange gültig, wie alle ihre Basisinstanzen existieren. Wird eine der eingehenden Basisinstanzen gelöscht, müssen auch die Structure-Instanzen der Association gelöscht werden, denen die gelöschte Basisinstanz zugrunde lag. Wenn eine Association bezüglich ihrer Basismengen vollständig dargestellt werden soll, können Instanzen der Association automatisch erzeugt werden, sowie sich die Basismengen verändern.

Transiente Attributes und Associations

Wenn die Funktionswerte einer Association berechnet werden können, ist es nicht immer sinnvoll, diese zu speichern, da auf diese Weise Redundanz erzeugt werden würde, die zusätzlicher Pflegemaßnahmen bedarf. In diesem Fall können die abgeleiteten Attributes als transiente Attributes definiert und erst bei der Bereitstellung einer Association-Instanz zu berechnet werden. Dadurch entfällt die Notwendigkeit des Aktualisierens, wenn sich Basisinstanzen ändern. Besitzt eine Association nur transiente Attributes und ist ihre Domäne vollständig bezüglich ihrer Argumente, also gleich der Produktmenge der Basismengen, kann die Association als transiente Association definiert werden. In diesem Fall werden die Instanzen der Association erst zur Laufzeit gebildet, indem die entsprechenden Basisinstanzen bereitgestellt und die Funktionswerte berechnet werden. Dieses Verfahren ist geringfügig langsamer als der Zugriff auf gespeicherte Zusammenhänge, hat aber den Vorteil eines erheblich reduzierten Wartungs- und Speicheraufwands.

Views Templates

Eine spezielle Form der Associations sind Views. Während in der Association die Basisinstanzen integraler Bestandteil der Association Instanz sind, wird im Rahmen einer View eine Auswahl der in der View darzustellenden Properties getroffen

wird. Außerdem kann, wie in der SQL, die Domäne einer View über Pre- und Post-Selektionen eingeschränkt werden.

Views Templates, hier kurz Views, stellen sowohl einen strukturellen als auch einen methodischen Sachverhalt dar. Der Strukturelle Aspekt besteht in der Structure der View und den Structures, auf denen diese View basiert. Der Methodische Aspekt beschreibt die Vorschriften zur Auswahl der Instanzen und zur Berechnung abgeleiteter Properties. So erfolgt die Darstellung einer View zwar im Strukturmodell. Da jedoch die Bildung der Property-Instanzen der View im wesentlichen auf Expressions beruht, ist die View sehr eng mit den entsprechenden Methoden verbunden (➔**SDB_View**)

Die Erzeugung einer View kann in den hier beschriebenen Schritten als Prozeß aufgefaßt werden. Dabei durchläuft die Erstellung einer View drei Phasen, die auch im Strukturmodell dargestellt werden:

- **View-Domäne**

  Die View-Domäne ist eine Association, die strukturell aus den Base Structures der View und ihren definierten Properties besteht. Die Menge der Instanzen der View-Domäne ist eine Teilmenge der Produktmenge der Base Structure Collections unter Berücksichtigung der Basisselektionen.

- **Elementare View**

  Die elementare View ist strukturell durch die Properties der View-Definition festgelegt. Mengenmäßig entspricht sie den Instanzen der View-Domäne, die der Pre-Selektion genügen.

- **Aggregierte View**

  Die aggregierte View enthält als Properties die Key Components des Gruppierungsschlüssels und die aggregierten Properties. Außerdem besitzt jede Instanz der aggregierten View eine Collection Reference **_partition**, die alle Instanzen der elementaren View mit dem Gruppierungsschlüssel der Instanz enthält. Mengenmäßig enthält die aggregierte View eine Instanz je Gruppierungsschlüssel-Instanz abzüglich der Instanzen, die ggf. die Bedingung der Post-Selektion nicht erfüllen.

Eine View wird in diesen drei Phasen erstellt, wobei anschließend sowohl die elementare als auch die aggregierte View verfügbar ist. Jede dieser Phasen ist mit einer entsprechenden Structure Definition verbunden.

Die View-Domäne

Eine View ist auf einer Menge von Instanzenmengen definiert. Dies können Extents oder durch Views definierte Mengen sein. Außerdem können in Views auch Reference Collections der Argument Structures einbezogen werden. Die Types dieser Mengen fungieren als Argument-Structures der View-Domäne.

Beispiel Um die Summe des Einkommens aller **Kinder** für eine PERSON darzustellen, muß die Menge der **Kinder** als Basismenge in die Bildung der View einbezogen werden.

```
View PersonView {
   view_domain PersonDomaene {
      baseStruct (
         M_Personen { extent Person;
                      ddeinit { Name == „Müller" } }
         M_Kinder     extent Kinder; )
   ...           }
}
```

Durch die **ddeinit**-Action werden aus dem Extent nur die PERSONEN mit dem **Namen** *Müller* bereitgestellt. **Kinder** bezieht sich auf die Menge der **Kinder**, die in der BaseStructure PERSON als Property definiert sind. PERSONEN ohne **Kinder** werden in diesem Fall nicht in die Sicht einbezogen.

Als Basismengen können sowohl freie Collections (Extents oder Views) als auch References aus Structures in Base Collections angegeben werden. Jede der Base Collections kann mit einer Selektionsbedingung versehen werden. Dadurch ist es möglich, die Menge auf niedrigster Ebene einzuschränken. Die Menge der auf diese Weise gebildeten Instanzen heißt View-Domäne. Die View-Domäne ist eine Association, die aus den Instanzen ihrer Base Structures besteht (➔**SDB_ViewDomain**)

Die elementare View

Für die elementare View können verschiedene Properties definiert werden. Dieses können entweder Properties der Base Structures oder abgeleitete Properties sein. In die Berechnung abgeleiteter Properties können nur Properties der Basisinstanzen referenziert werden.

Beispiel Um die Summe des Einkommens der **Kinder** in der View zu definieren, kann eine Berechnungs-Expression oder eine Funktion definiert werden. Andere Properties wie Name oder Vorname können direkt übernommen werden.

```
View PersonView {
   view_domain PersonDomaene {
      baseStruct (
         M_Personen { extent Person;
                      ddeinit { Name == „Müller“ } }
         M_Kinder     extent Kinder; )
   }
   view_elementary PersonElements {
      attribute(
         P_Name      ddeinit M_Personen.Name;
         J_Einkommen ddeinit M_Kinder.Jahreseinkommen()
         P_Vorname   ddeinit M_Personen.Vorname;        )
      pre_selection  CheckEinkommen;
   }
}
```

J_Einkommen ist das berechnete jährliche Einkommen eines Kindes. **P_Name** und **P_Vorname** werden direkt mit den Attributes der Basisinstanz en belegt.

Properties der elementaren View können neben Attributes auch References oder Relationships. Eine weitere Einschränkung der elementaren View, die der **WHERE**-Klausel in der SQL entspricht, ist die Pre-Selektion. Diese wird als Action angegeben und führt eine Auswahl der Instanzen vor der Aggregation durch. In die Pre-Selektion können die Properties der elementaren View einbezogen werden. Im Ergebnis der Pre-Selektion entsteht die elementare View, die Instanzen mit den abgeleiteten Properties enthält. Die Instanzen der Base Structures sind nicht Bestandteil der elementaren View. Durch die Definition von Keys und Indizes können verschiedene Sortierfolgen für die elementare View festgelegt werden (➔**SDB_ViewElement**).

Die aggregierte View

Views können, müssen jedoch nicht aggregiert werden. Aggregierte Views werden durch die Bildung von Partitions erzeugt. Das Gruppieren oder Bilden von Partitions erfolgt über einen Gruppierungsschlüssel, der als Ident Key der View Structure definiert wird. Alle Instanzen der elementaren View mit gleichem Wert in ihrer Gruppierungsschlüssel-Instanz werden zu einer Partition zusammengefaßt und als Collection Reference der View-Instanz abgelegt. Die Instanzen der Partition werden zu einer View-Instanz aggregiert. Der Gruppierungsschlüssel stellt den identifizierenden Schlüssel für die Instanzen der aggregierten View dar.

Beispiel Da die View auf der Personenebene definiert ist, wird für jede PERSON eine Partition gebildet, auf deren Ebene die aggregierten Daten wie die Summe des Einkommens der Kinder berechnet werden.

```
View PersonView {
   view_domain PersonDomaene {
      baseStruct (
         M_Personen { extent Person;
                      ddeinit { Name == „Müller“ } }
         M_Kinder     extent Kinder; )
   }
   view_elementary PersonElements {
      attribute (
         P_Name      ddeinit M_Personen.Name;
         J_Einkommen ddeinit M_Kinder.Jahreseinkommen()
         P_Vorname   ddeinit M_Personen.Vorname;      )
      pre_selection  CheckJEinkommen;
   }
   aggregate YES;
   attribute  S_Einkommen ddeinit sum_Einkommen;
   ident_key  group_key components( P_Name; P_Vorname );
   post_selection  CheckSumEinkom;
}
```

Die aggregierte View enthält für jeden **P_Namen/P_Vornamen** eine Instanz, in der eine Reference **_partition** definiert ist, die alle Instanzen der elementaren View mit gleichem Schlüsselwert enthält. Aggregationsfunktionen wie **sum()** sind Funktionen der System Class CollectInstance und können für jede Collection ausgeführt werden. Hier wird als Parameter der Name der zu aggregierenden Property mitgegeben.

Die aggregierte View enthält vorerst je eine Instanz pro Gruppierungsschlüssel-Instanz. Neben den Attributes des Gruppierungsschlüssels und der Collection Reference **_partition** können weitere Properties als Attributes, References oder Relationships für die aggregierte View definiert werden.

Durch die Post-Selektion ist eine weitere Einschränkung der Menge der Instanzen in der View möglich. In die Post-Selektion können nur Properties der View einbezogen werden. Die Instanzen der elementaren View sind für aggregierte Views über die **_partition** References verfügbar. Die aggregierte View enthält jetzt nur noch Instanzen, die der Bedingung der Post-Selektion entsprechen (➔**SDB_View**).

3.2 Collections für Mengen

Während Structures Mengen von benannten Instanzen definieren, können unbenannte Instanzen in Collections zusammengefaßt werden. Da Collections selbst wieder Instanzen sind, können sie z.B. wieder als benannte Property-Instanz in einer Structure definiert sein. In ODABA2 werden Collections als spezielle References aufgefaßt. Eine Reference ist dabei ein generischer Type, der auf eine Menge von Structure-Instanzen verweist. Diese Menge kann im speziellen Fall auch aus einer Instanz bestehen. Können in einer Reference mehrere Structure-Instanzen referenziert werden, wird sie als Collection Reference oder kurz als Collection bezeichnet. Es gibt jedoch auch freie Collections, die als Extents bezeichnet werden.

Identifizierung von Instanzen in Collections

Da Instanzen in Collections i.allg. keine benannten Instanzen sind, müssen sie in geeigneter Weise angesprochen werden können. Die Instanzen-Identity ist dabei nicht immer geeignet. Structure-Instanzen in Collections können auf dreierlei Weise identifiziert werden:

- **Über Schlüssel eines Indexes**

  Eindeutige Schlüssel sind Schlüssel, die innerhalb eines Indexes eindeutig sind. Für einen Extent können neben dem Ident Key andere eindeutige Sortierschlüssel definiert werden, die zum gezielten Zugriff auf die Instanzen einer Collection verwendet werden können.

- **Über die Position**

  Dem Zugriff von Instanzen über Position liegt immer ein Index zugrunde. Die Position gibt die Position einer Instanz im eingestellten Index an. Auch für unsortierte Collections wird ein Index mit den Instanzen-Identities angelegt, nur daß dieser nicht sortiert ist. Der Zugriff über Position ist allerdings problematisch, da bei paralleler Verarbeitung Verschiebungen durch Löschen oder Hinzufügen in den Indizes auftreten können.

- **Über die Instanzen-Identity**

  Der Zugriff über die Instanzen-Identity ist möglich, wenn ein Identity-Index mit dem Schlüssel __IDENTITY für die Collection definiert wurde. Identity-Indizes müssen jedoch nicht unbedingt eindeutig (unique) sein. Ein eindeutiger Zugriff über Instanzen-Identity ist in einer Collection also nur möglich, wenn diese einen eindeutigen Identity-Index besitzt. Da die Identity jedoch in jedem Fall eine bestimmte Instanz bezeichnet, bestehen die Unterschiede für nicht eindeutige Identity-Indizes nur in der eingestellten Position des Indexes.

3.2.1 References

References sind Bezüge auf einzelne oder Mengen von Structure-Instanzen. References treten sowohl in Structure-Definitionen als Properties (References und Relationships) als auch in RealObjects als Extents auf. References stellen Verweise auf Structure- oder Collection-Instanzen dar (➔**SDB_Reference**). Structure-Instanzen sind generell nur über References erreichbar, d.h. sie werden von mindestens einer Reference referenziert. Es ist jedoch möglich, Structure-Instanzen von beliebig vielen References zu referenzieren.

Owning References

Da eine Structure-Instanz in mehreren References referenziert werden kann, besteht die Möglichkeit, eine Reference als Besitzer ihrer Structure-Instanzen zu definieren. References, die ihre referenzierten Instanzen besitzen, werden Owning References genannt. Structure-Instanzen einer Owning Reference sind hinsichtlich ihrer Lebensdauer an die Lebensdauer der Reference gebunden. Owning References sind also in dem Sinne instanzenerzeugend, als daß Instanzen beim Hinzufügen (z.B. über IdentKey) automatisch erzeugt werden. Ebenso werden sie beim Entfernen aus der Owning Reference gelöscht.

Das Hinzufügen einer Instanz zu einer (not Owning) Reference erfolgt durch das direkte oder indirekte Hinzufügen einer existierenden Instanz Identity oder anhand des Identschlüssels. Beim Entfernen einer Instanz aus einer (not Owning) Reference wird nur die Instanzen-Identity aus der Reference entfernt, während die Instanz selbst erhalten bleibt, bis sie aus der Owning Reference gelöscht wird. Da die gelöschte Instanz in anderen (not Owning) References noch referenziert werden kann, kann dies später zu Fehlern führen. Inkonsistenzen dieser Art können durch die Verwendung von Controlled References vermieden werden. Treten sie jedoch auf, reagiert ODABA2 mit einem definierten Zugriffsfehler.

Transiente und persistente References

References können als transiente oder Persistente References definiert werden. Während persistente References in der Datenbank gespeichert werden, sind transiente References nur im Rahmen eines laufenden Prozesses verfügbar. Sie entstehen häufig als Ergebnis von Mengenoperationen.

Types und References

References können auf unterschiedliche Weise mit einem Type verbunden sein, der Festlegungen zu den Types der Instanzen der Reference trifft.

- **Typed Reference**

  Wenn eine Menge nur Instanzen eines Types enthalten darf, wird sie als Typed Reference bezeichnet. Die meisten References sind Typed References.

- **Untyped References**

  References, die Instanzen beliebigen Types enthalten können, werden Untyped References genannt. Untyped References ermitteln den Type der Instanzen zum Zeitpunkt der Bereitstellung der Daten. Das setzt voraus, daß der Type beim Erzeugen der Instanzen bekannt ist. Die Bildung von Indizes ist für Instanzenmengen ist in diesem Fall nicht möglich.

  Untyped References können entweder Instanzen unterschiedlichen Types oder Instanzen gleichen Types enthalten.

  Beispiel In einem geordneten Lager wird es BEHÄLTER geben, die zwar alles mögliche, jedoch immer nur Dinge einer bestimmten Art enthalten. BEHÄLTER hat also eine multiple Untyped Reference für seinen **Inhalt**. Die Art des Inhaltes wird spätestens dann festgelegt, wenn das erste Ding in den Behälter getan wird. Der BEHÄLTER bekommt ein Schild, das ihm i.allg. solange anhaften bleibt, bis der BEHÄLTER wieder leer ist. Dann kann er wieder für andersartige Dinge benutzt werden.

  In diesem Fall kann die Reference mit dem Type VOID (Untyped) definiert werden. In dem Moment, wo die erste Instanz zugeordnet wird, wird der Type der Reference für die aktuelle Instanz festgeschrieben, so daß nun nur noch Instanzen des gleichen Types zu der Reference hinzugefügt werden können. Erst wenn die Reference leer wird, kann der Type geändert werden.

- **Weak typed References**

  Zwischen diesen beiden Extremen liegen References, deren Instanzen die gleiche Base Structure besitzen. Diese References werden als **Weak Typed References** bezeichnet.

  Soll eine Reference Instanzen undefinierten Types unterschiedlicher Art enthalten können (z.B. eine Garbage Collection, in der aller Müll gesammelt wird), muß die Reference als VOID (Untyped) und Weak Typed definiert werden.

Enum-basierte References

Enum-basierte References sind kontrollierte Untyped oder Weak Typed References in Structure-Instanzen, deren Type durch ein Attribute mit einer Extended Enumeration in der gleichen Structure-Instanz festgelegt wird. Die Instanz enthält also die Informationen über den Type der Reference.

Beispiel Eine Person kann entweder Abiturient oder Fachschulabsolvent usw. sein (siehe Beispiel in 3.1.2 zu „Extended Enumerations“). Wird also in der Structure PERSON das Attribute **Abgang** wie folgt definiert, kann es als Metainformation zur Bildung einer entsprechenden Ableitung der Instanz dienen:

```
structure Person {
   attribute (
      Name       { type STRING; size 30 }
      Vorname    { type STRING; size 30 }
      Abgang     { type Abschluss }              )
   relationship (    Name       { type STRING; size 30 }
                     Vorname    { type STRING; size 30 }
                     Abgang { type Abschluss }    )
}
```

Controlled References

Da Structure-Instanzen in verschiedenen References referenziert werden können, muß die Konsistenz dieser References gesichert werden, d.h. gelöschte Instanzen müssen aus den entsprechenden References entfernt und indizierte Collections bezüglich ihrer Indizes bei Instanzenänderungen aktualisiert werden. Da diese Maßnahmen aufwendig und nicht immer erwünscht sind, können References, deren Konsistenz gesichert werden soll, explizit als Controlled References markiert werden.

Geteilt benutzbare References

In geteilt benutzbaren References gespeicherte Instanzen können durch mehrere Anwendungen gleichzeitig bearbeitet werden. Da in diesem Fall spezielle Speicherformen und Locking-Strategien erforderlich sind, sind geteilt benutzbare

References in der Verarbeitung langsamer. Das betrifft auch Single User-Anwendungen. Extents sind in jedem Fall geteilt benutzbar. Andere References wie z.B. Relationships mit inversen References der Extents verlangen ausdrücklich geteilte Benutzbarkeit.

3.2.2 Collections

Collections sind spezielle References, die Instanzenmengen verwalten. Eine Collection wird als Collection-Instanz gespeichert und kann auch als solche referenziert werden. In einer Collection kann die Eindeutigkeit der Instanzen bezüglich der Instanzen-Identity gefordert werden. Ist eine Collection nicht als unique gekennzeichnet, können Instanzen mehrfach in der Collection auftreten. Diese Eigenschaft hat nichts mit der Eindeutigkeit ggf. definierter Indizes gemeinsam, über die zusätzliche Eindeutigkeitskriterien festgelegt werden (➔**SDB_Reference**).

3.2.2.1 Collection-Kategorien

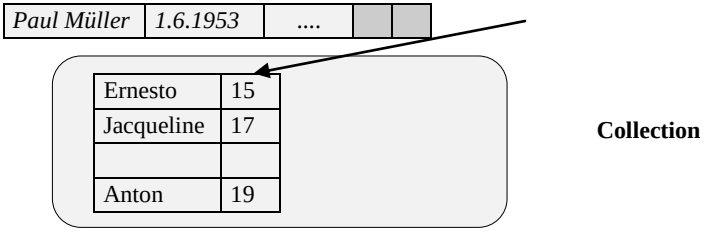
Entsprechend der Speicherform werden drei Kategorien für Collections unterschieden. Die Kategorie einer Collection ergibt sich i.allg. aus dem Zusammenhang, in dem die Collection definiert wird und wird nicht explizit angegeben.

Dynamic Arrays

Dynamic Arrays sind Collections, in denen die Structure-Instanzen direkt abgelegt werden. Instanzen in Dynamic Arrays sind keine freien Instanzen und erhalten somit auch keine Instanzen-Identity. Dynamic Arrays sind Owning References, deren Instanzen nicht in anderen References referenziert werden können. Aufgrund der direkten Speicherung sind Dynamic Arrays die zeitsparendste Variante der Speicherung von Collections, allerdings auch die mit den meisten Einschränkungen. Auf Dynamic Arrays wird über die Position im Bereich zugegriffen. Die Bildung von Indizes über Dynamic Arrays wird nicht unterstützt.

Wird ein Element an das Ende des Arrays angefügt, werden ggf. nötige leere Instanzen automatisch erzeugt.

Beispiel Wenn die **Kinder** einer PERSON als Dynamic Array dargestellt werden sollen, wird folgende Collection-Instanz erzeugt:

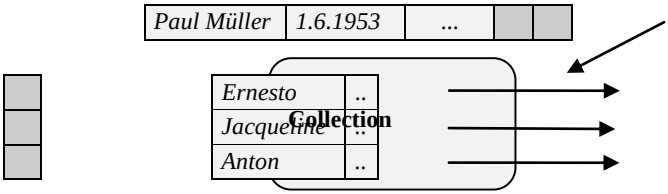


Da *Anton* dem Dynamic Array als viertes Kind hinzugefügt wurde, ist eine leere Instanz als dritte Instanz des Arrays hinzugefügt worden.

Einfache Collections

Die zweite Möglichkeit der Speicherung von Collection-Instanzen besteht in der Speicherung einfacher Collections. In diesem Fall wird statt der Instanzen eine Liste von Identities der Instanzen der Collection, ein Index, gespeichert. Die Instanzen der Collection können dabei auch in anderen References referenziert werden.

Beispiel Wenn die **Kinder** einer PERSON als einfache Collection gespeichert werden, enthält die Collection eine Liste von Identities für die Kinder-Instanzen.

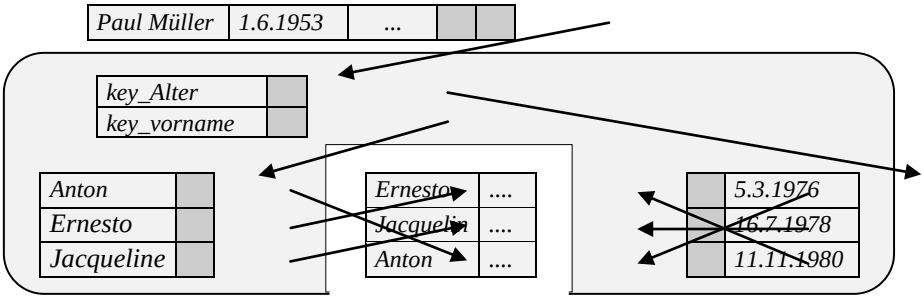


Die Nachteile beim Zugriff können dadurch ausgeglichen werden, daß die Instanzen in unmittelbarer Nachbarschaft, in einem Cluster, gespeichert werden. Anstelle der unsortierten Identity-Liste kann auch eine Indizierte Liste erzeugt werden. In diesem Fall werden die Identities in der Collection entsprechend dem angegebenen Schlüssel sortiert.

mehrfach indizierte Collections

Mehrfach indizierte Collections stellen die leistungsfähigste, aber auch die aufwendigste Variante der Speicherung von Instanzenmengen dar. Da Indizes über Schlüssel der referenzierten Structure-Instanzen gebildet werden, sind sie vom Zustand der Instanzen abhängig und müssen bei jeder Änderung einer Instanz aktualisiert werden.

Beispiel Wenn die **Kinder** einer PERSON als mehrfach indizierte Collection gespeichert werden, können Indizes über mehrere Schlüssel gebildet werden, über die jeweils ein sortierter Zugriff auf die Collection stattfinden kann.



Durch Auswahl eines Indexes *key_Alter* oder *key_vorname* erfolgt der Zugriff auf die Instanzen in der Folge des **Geburtsdatums** oder des **Vornamens**. Beim Ändern einer PERSONEN-Instanz der **Kinder** müssen in diesem Fall alle betroffenen Indizes aktualisiert werden.

Die Konsistenzsicherung der Indizes wird in ODABA2 nur für Controlled References gewährleistet.

3.2.2.2 Indizes

Indizes werden verwendet, um Mengen in Extents und Referenzen zu verwalten. Indizes können nur für typed oder Weak Typed Collections definiert werden. Im Gegensatz zum OMDG-93 Standard sind Eigenschaften wie Eindeutigkeit, Sortierfolge usw. in ODABA2 nicht Eigenschaften der Collection, sondern Eigenschaften eines Indexes und können zwischen den verschiedenen Indizes eines Extents variieren. Alle Indizes werden auf der Basis von Keys definiert, die in der

Structure festgelegt werden. Eine Ausnahme bildet der Identity Key __IDENTITY, der nicht explizit definiert werden muß (➔**SDB_Index**).

Die Indexdefinition beschreibt Größe und Typ der zu verwaltenden Menge. Es werden u.a. die Standard-Collections **Bag** (nicht eindeutig, unsortiert), **Set** (nicht eindeutig, sortiert) und **List** (eindeutig, sortiert) unterstützt. Die Kombination der verschiedenen Angaben erlaubt jedoch darüber hinaus weitere Arten von Collections. Die Eindeutigkeit in einem Index wird hinsichtlich des Sortierschlüssels festgestellt. Insofern ist die Eindeutigkeitsforderung schärfer, als in der **Liste** gefordert, da auch verschiedene Instanzen aufgrund gleicher Schlüssel ausgeschlossen werden können.

maximale Instanzenzahl

Dieser Wert dient der Ermittlung einer optimalen Indexstrategie. Normalerweise ergibt er sich aus der Dimensionsangabe in der Reference. Da für Mehrfachschlüssel die Kardinalität des Indexes jedoch von der Kardinalität der verwalteten Menge abweichen kann, kann hier ein maximaler Wert festgelegt werden, der die Anzahl der Index-Einträge begrenzt. In Abhängigkeit von diesem Wert werden optimale Indexstrategien ermittelt.

Eindeutige Indizes

Hinsichtlich der Schlüsselwerte kann Eindeutigkeit in einem Index gefordert werden. Die Eindeutigkeitsbedingung in einer Collection Extent führt dazu, daß Instanzen nur aufgenommen werden, wenn der hinzuzufügende Schlüsselwert in dem Index noch nicht existiert. Diese Bedingung wird für aller eindeutigen Schlüssel einer Collection geprüft.

Temporärer Index

Dieses Feld zeigt an, daß die Sortierfolge nur temporär erzeugt werden soll. Das kann für kleine Mengen ebenso sinnvoll sein wie für abgeleitete Extents.

3.2.3 Extents

Extents sind Collections, die außerhalb von Structure-Instanz existieren. In ODABA2 sind Extents Properties von RealObjects. In diesem Sinne stellen Extents benannte Instanzen in einem RealObject dar, deren Namen innerhalb der RealObjects eindeutig sein müssen. Welche Extents einem RealObject zugeordnet werden, wird jedoch nicht auf der Schemaebene, sondern erst auf der Sachebene festgelegt. Extents sind immer mit einem Structure Type verbunden und eindeutig nach dem IdentKey ihrer Structure sortiert. Dabei können für jedes RealObject eine Menge von Extents für jeden Structure Type gebildet werden.

IdentKey

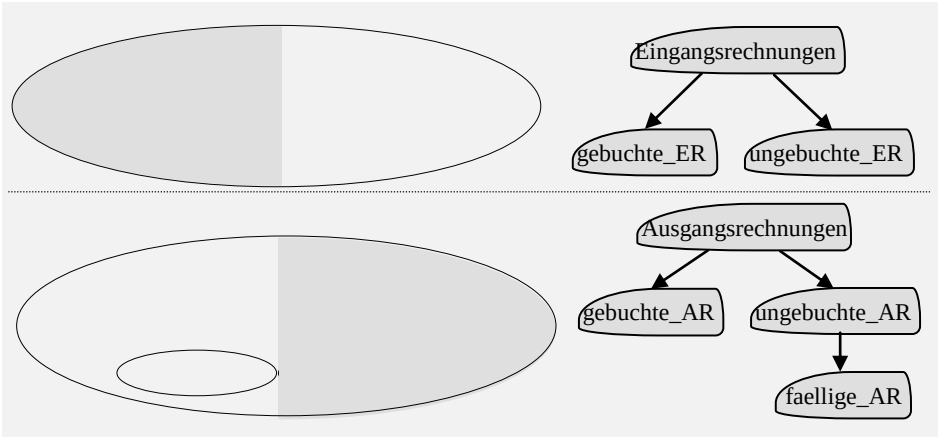
Da die Verwendung der Instanzen-Identity zur Kommunikation nach außen nicht besonders geeignet ist, werden die Extents über IdentKeys verwaltet. Jedem Extent liegt also mindestens der IdentKey Index zugrunde. Weitere Indizes können definiert werden. Ähnlich wie die Instanzen-Identity sind IdentKey Instanzen innerhalb eines Extents eindeutig. Damit kann z.B. durch Angabe des IdentKey eine Objektbeziehung hergestellt werden, was auf der Anwendungsebene wesentlich einfacher ist als die Verwendung der Instanzen-Identity.

Extent-Hierarchien

Jeder Extent kann eine oder mehrere Base Extents besitzen, die die Obermengen des Extents präsentieren. Extents, die keine Base Extents, als keine Obermengen besitzen, heißen RootExtent. RootExtents sind i.allg. Owning Collections, während abgeleitete Extents nie als Owning Collections definiert werden können. Zu einem Type können mehrere RootExtents definiert werden. Die RootExtents eines Types sind als Owning Collections zwangsläufig disjunkt zueinander.

Da ein abgeleiteter Extent Teilmenge jedes Base Extents ist, stellt er eine Teilmenge des Durchschnitts aller seiner Base Extents dar. Jeder RootExtent stellt die Spitze einer Hierarchie abgeleiteter Extents dar, die als abgeleitete Extents bezeichnet werden. Zwischen diesen können verschiedene Teilmengenbeziehungen definiert werden.

Beispiel In der Buchhaltung eines Unternehmens spielen RECHNUNGEN eine Rolle. Dabei wird zwischen **Eingangsrechnungen** und **Ausgangsrechnungen** unterschieden. Die Bezeichnung **Eingangs-** und **Ausgangsrechnung** bringt dabei eine bestimmte Rolle der RECHNUNG zum Ausdruck, da eine **Eingangsrechnung** für ein Unternehmen die **Ausgangsrechnung** für ein anderes Unternehmen ist. In jedem Fall handelt es sich dabei um RECHNUNGEN, d.h. der Type ist in jedem Fall RECHNUNG. Da es sich andrerseits um zwei völlig getrennte Mengen von RECHNUNGEN handelt, die in keinerlei Mengenbeziehung zueinander stehen, scheint es angemessen, jeweils einen RootExtent für **Eingangsrechnungen** und einen für **Ausgangsrechnungen** zu definieren. In diesen Mengen sind dann wieder gebuchte und nicht gebuchte RECHNUNGEN von Interesse. Bei **gebuchten Ausgangsrechnungen** könnten wiederum **fällige Rechnungen** eine besondere Rolle spielen.



Dieser Zusammenhang kann durch folgende Extent-Definitionen dargestellt werden:

```
Extent  Eingangsrechnung {type Rechnung; disjunct YES;
                          union YES }
Extent  gebuchte_ER       based_on Eingangsrechnung;
Extent  ungebuchte_ER     based_on Eingangsrechnung;

Extent  Ausgangsrechnung {type Rechnung; disjunct YES;
                          union YES}
Extent  gebuchte_AR       based_on Ausgangsrechnung;
Extent  ungebuchte_AR     based_on Ausgangsrechnung;
Extent  faellige_AR       based_on ungebuchte_AR;
```

Extent-Hierarchien bilden Teilmengenbeziehungen zwischen Controlled Collections ab. Für eine Menge $\mathbf{M_0}$ in einer Extent-Hierarchie gelten folgende

Beziehungen zu ihren Base Extents $\mathbf{B_1}$ ... $\mathbf{B_n}$ und ihren abgeleiteten Extents $\mathbf{D_1}$ ... $\mathbf{D_m}$:

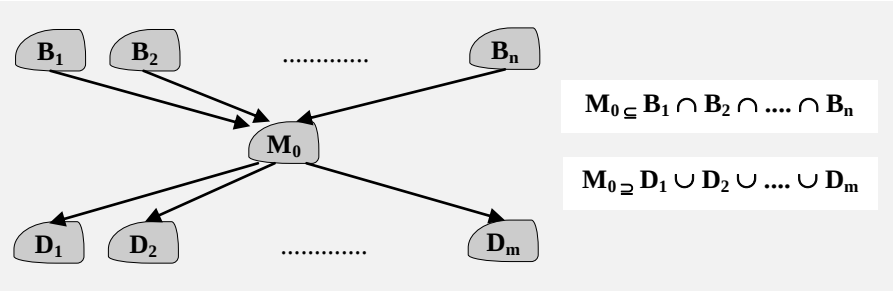


Abbildung 4.1: Mengenbeziehungen in Extent-Hierarchien

Da jeder Extent zum einen die Teilmenge des Durchschnitts seiner Base Extents ist und zum anderen eine Obermenge der Vereinigung der abgeleiteten Extents darstellt, können die wesentlichen Anforderungen der elementaren Mengenalgebra innerhalb von Extent-Hierarchien durch wenige Eigenschaften abgebildet werden.

- **Disjunkte Teilmengen**

  Die Zerlegung eines Extents in abgeleitete Extents ist disjunkt, wenn eine Instanz des Base Extents in höchstens einem der abgeleiteten Extents existiert.

- **Union (echte Vereinigungsmenge)**

  Ein Extent ist die Union seiner abgeleiteten Extents, wenn jede Instanz des Extents in wenigstens einem der abgeleiteten Extents existiert. Ist ein Extent die Union seiner abgeleiteten Extents, sprechen wir auch von einer vollständigen Zerlegung in abgeleiteten Extents. Ist die Zerlegung eines Extents vollständig und disjunkt, existiert jede Instanz in genau einem abgeleiteten Extent. Das entspricht dem Konzept der Systematiken und Kategorien auf der Mengenebene. Eine vollständige und disjunkte Zerlegung in zwei Mengen bildet eine Menge und ihre Komplementär- oder Differenzmenge ab.

- **Intersection (echte Durchschnittsmenge)**

  Ein Extent ist der echte Durchschnitt seiner Base Extents, wenn jede Instanz, die in allen Base Extents enthalten ist, Instanz des Extents ist.

Controlled Extents

Extents sind i.allg. als Controlled Extents definiert. Für Controlled Extents werden die Mengenbeziehungen der Extent-Hierarchie auch beim Löschen und Hinzufügen von Instanzen gesichert. Beim Hinzufügen eines Elementes zur Menge wird die Instanz auch in allen Basismengen hinzugefügt (falls noch nicht vorhanden). Beim Löschen einer Instanz aus einer der Basismengen wird die Instanz auch aus dem abgeleiteten Extent entfernt. Außerdem werden für Controlled Extents bei Änderungen der Instanzen die Indizes aller Extents der Hierarchie gepflegt.

3.3 Properties zur Darstellung von Merkmalen

Die Properties einer Structure bilden die Merkmale einer Objektsicht oder eines funktionalen Zusammenhangs ab. Properties können einzelne Werte, Structure-Instanzen oder Collections von Werten oder Structure-Instanzen darstellen. Properties werden als eingebettete Instanzen oder als Referenzen auf Instanzen in der Structure-Instanz gespeichert. Eine Property stellt sowohl ein Merkmal (➔**SDB_Property**) als auch kausalen Zusammenhang (➔**CAU_Causality**) dar. Kausale Zusammenhänge werden später im Zustandsmodell beschrieben.

Properties können wie die Merkmale im OSM nach formalen und inhaltlichen Gesichtspunkten systematisiert werden.

3.2.1 Formale Kategorien für Properties

So wie die Merkmale einer Sicht im OSM in feste, variable und undefinierte Merkmale eingeteilt wurden, ergeben sich auch für die Properties verschiedene formale Kategorien.

- **Static Property**

  Statische Properties bilden die festen Merkmale einer Sicht ab. Feste Merkmale sind Merkmale, deren Wert für alle Instanzen einer Art gleich sind (z.B. die Anzahl der Seiten für Vierecke).

- **Variable Property**

  Variable Properties werden als Properties im Kontext der Structure abgebildet. Sie stellen die variablen Merkmale einer Sicht dar.

- **Undefined Property**

  Undefined Properties bilden freie Merkmale einer Sicht ab. Da auf der Schemaebene nicht bekannt ist, welche Merkmale auf diese Weise abgebildet werden, können sie als solche auf der Schemaebene auch nicht definiert werden. Extents für RealObjects sind ein Beispiel für undefined Properties.

Obwohl sowohl statische als auch variable Properties dargestellt werden können, entspricht die derzeitige Form nicht dem Konzept, das diesem Sachverhalt zugrunde liegt. Dieses bleibt der Weiterentwicklung von ODABA2 vorbehalten.

3.2.2 Eigenschaften von Properties

Properties sind Member einer Structure (➔**SDB_Member**), die im Rahmen einer Structure Instanz gespeichert werden können. Neben ihren Member-Eigenschaften besitzen Properties jedoch einige besondere Eigenschaften.

Properties können als transiente Properties definiert werden. Transiente Properties sind Properties einer Structure, die nicht gespeichert werden. Transiente Properties werden z.B. verwendet, um abgeleitete Merkmale zu berechnen und intern in der Structure-Instanz bereitzustellen. Sie existieren nur in der internen Darstellung einer Structure-Instanz im Rahmen einer Anwendung.

Als Member können eine Reihe weiterer Eigenschaften für Properties definiert werden. Diese Eigenschaften haben Properties z.B. mit Methoden gemeinsam, die Ebenfalls spezielle Member einer Class, also einer speziellen Structure, sind. Member sind i.allg. im Kontext einer Structure oder einer Class definiert.

- **Name**

  Namen sind case-sensitiv. Sie müssen im Kontext, in dem sie als Member definiert werden, eindeutig sein.

- **Type**

  Der Type legt die Art der Darstellung der Property fest. Untyped Properties werden mit der Type-Angabe VOID definiert. Untyped Properties können nur als transiente Zeiger oder References definiert werden. Als Types können für Zeiger und References auch Type-Deklarationen oder unvollständige Type-Definitionen angegeben werden. Allerdings können Instanzen einer Reference erst referenziert werden, wenn die Definition vollständig ist.

- **Generic Type**

  Structure Types können mit generischen Types assoziiert werden, wenn damit eine bestimmte Verwaltungsform der Instanzen dieses Types assoziiert werden soll. Als generische Typen können nur Typen angegeben werden, die ausdrücklich als generische Typen definiert wurden.

- **Size**

  Die Größe des Members bestimmt die Menge des zu reservierenden Speicherplatzes. Sie legt fest, wie viele signifikante Zeichen oder Ziffern in dem Member abgelegt werden können.

- **Precision**

  Die Angabe der Genauigkeit hat i.allg. keinen Einfluß auf die Speicherung der Werte. Sie wird jedoch an bei Datenkonvertierungen zur Berechnung und zur Darstellung der Werte im Zeichenformat verwendet. Die Regeln sind für die verschiedenen Datentypen unterschiedlich und im Zusammenhang mit dem Datentyp beschrieben (**➔DataTypes**).

- **Dimension**

  Die Angabe einer Dimension kann die maximale Anzahl von Instanzen einer Property begrenzen. Für Attribute Arrays gibt sie die Anzahl der anzulegenden Bereichselemente an. Für Collections bestimmt sie die maximale Kardinalität der Instanzenmenge, die in der Collection abgelegt werden kann.

- **Referenzlevel**

  Das Referenzlevel definiert, ob ein Feld direkt oder über Zeiger (Referenzlevel > 0) angesprochen wird. Zeiger können nur für transiente Felder oder in transienten Strukturen definiert werden. Auf diese Weise können auch transiente References definiert werden.

- **Zugriffsrechte**

  Zugriffsrechte für Properties legen fest, aus welchen Methoden auf Properties zugegriffen werden darf. Diese Angabe spielt vor allem im Zusammenhang mit der Spezialisierung zu Classes eine Rolle.

- **Static Properties**

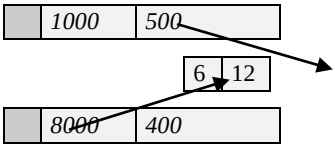
  ODABA2 unterstützt Static Properties im Sinne von C++, d.h. als unveränderliche Merkmale eines Types. Nicht unterstützt wird gegenwärtig der Wechsel dieser Eigenschaft innerhalb einer Type-Hierarchie, d.h. eine einmal als Static definierte Property ist auf allen Ableitungsebenen mit dieser Eigenschaft verbunden. Es ist also derzeit nicht möglich, die Anzahl der **Ecken** für ein VIERECK als Static zu definieren, während sie für die BaseStructure POLYGON nicht fest, also eine Variable Property ist.

  Static Properties werden in einer allen Instanzen gemeinsamen Sub-Instanz gespeichert, wie das folgende Beispiel zeigt.

Beispiel 4.18 Die Anzahl der **Flächen** und **Kanten** sind für alle WÜRFEL gleich. Es handelt sich um feste Merkmale, die als Static Property dargestellt werden können

```
structure Würfel {
   attribute (  Volumen     { type INT; size 10 }
                Gewicht     { type INT; size 10 }
                Flächen     { type INT; static YES}
                Kanten      { type INT; static YES}    )
}
```

Für die Instanzen für WÜRFEL würde sich dann folgende Darstellung ergeben:



- **Virtual Property**

  In vielen Zusammenhängen findet mit der Spezialisierung einer Sicht auch eine Spezialisierung der Merkmale statt, d.h. die Sicht auf die Bestandteile oder Eigenschaften eines Sachverhaltes verändert sich.

  Beispiel Bei der Darstellung eines ZUGES werden **Wagen** als Bestandteile des ZUGES abgebildet. Da ein PERSONENZUG ein spezieller ZUG ist, wird es sich bei den WAGEN auch um spezielle WAGEN handeln. So hat der PERSONENZUG PERSONENWAGEN, der GÜTERZUG GÜTERWAGEN. Das kann durch die Definition von Virtual Properties ausgedrückt werden:

```
structure Zug { ...
   reference  alleWagen {type Wagen; virtual YES}
   ...          }

structure Güterwagen {
   baseStruct Wagen      { type Wagen }
   ...          }

structure Güterzug {
   baseStruct Zug          type Zug;
   ...
   reference  alleWagen { type Güterwagen; virtual YES }
}
```

  Hier ist die Reference **alleWagen** für Güterzug überladen worden. Die einzige Voraussetzung dafür ist, daß die überladenen Properties den gleichen Namen wie in der abgeleiteten Structure haben und die Types der überladenen Properties Base Structures des überladenen Types sind., WAGEN also Base Structure von GÜTERWAGEN ist.

- **Initialisierungswert**

  Mittels einer Initialisierungswertes ist es möglich, einzelne Member zu initialisieren. Beim Erzeugen persistenter Instanzen werden diese Werte verwendet, um die Properties einer Instanz beim Konstruieren vorzubelegen.

3.2.3 Inhaltliche Kategorien für Properties

Aus den Merkmalstypen des Sprachmodells ergeben sich vier inhaltliche Kategorien für Properties.

- **Base Structure**

  Base Structures bilden die Basissichten einer Sicht ab. Sie stellen somit die hierarchischen Beziehungen zwischen Structure-Instanzen eines Objektes dar.

- **Attribute**

  Attributes bilden Eigenschaften eines Sachverhaltes im Rahmen einer Structure ab. Attributes sind als Eigenschaften fest mit ihrem Träger (der Structure-Instanz) verbunden.

- **Reference**

  References beschreiben Verweise auf andere Instanzen. Dabei kann es sich um einzelne Structure- oder um Collection-Instanzen handeln.

- **Relationship**

  Relationships beschreiben die Beziehung zwischen unabhängigen Objekten aus einer bestimmten Sicht. Vermittelt über die Structure-Instanzen werden somit Objektbeziehungen dargestellt. Dabei können im Rahmen einer Relationship verschiedene Abhängigkeiten definiert werden.

3.2.3.1 Attributes für Eigenschaften

Attributes bilden als Properties die Eigenschaften einer Objektsicht ab. In funktionalen Zusammenhängen werden die Funktionswerte über Attributes dargestellt. Attributes sind, wie die Eigenschaften, fest mit ihrem Träger verbunden. Jede Structure-Instanz enthält also Werte für alle Attributes der Structure. Eine Attribute-Instanz existiert nur in der Structure-Instanz. Sie wird mit der Structure-Instanz angelegt und verschwindet, wenn die entsprechende Structure-Instanz gelöscht wird. Dazwischen kann die Attribute-Instanz verschiedene Werte annehmen, kann aber nicht gelöscht werden.

Attributes sind immer mit einem Type verbunden, der den Aufbau des Attributes beschreibt. Dieser Type kann ein Literal zur Darstellung elementarer Arten, eine Enumeration zur Darstellung von Kategorien oder eine Structure zur Abbildung komplexer Eigenschaften sein. Werden Structure-Instanzen als Attribute-Instanzen erzeugt und abgelegt, wird ihnen keine Instanzen-Identity zugeordnet. Attribute-Instanzen sind in diesem Sinne keine freien Instanzen, sondern können nur im Kontext der Structure-Instanz erreicht werden.

Beispiel Die ANSCHRIFT ist eine Eigenschaft der PERSON, die fest mit ihr verbunden ist, d.h. jede PERSON hat irgendeine ANSCHRIFT (jedenfalls in einer idealen Gesellschaft), die strukturiert als ADRESSE abgelegt werden kann. Die **Anschrift** einer PERSON ist also eine Eigenschaft der Person und wird somit als Attribute abgebildet.

```
structure Person {
   attribute (  ...
                Anschrift { type Adresse }            )
   ....
}
structure Adresse {
   attribute (  Strasse    { type STRING; size 40 }
                Ort        { type STRING; size 30 }
                PLZ        { type STRING; size 5 }
                Land       { type CHAR; size 1 }     )
}
```

Die **Anschrift** der PERSON (ADRESSE) wird hier direkt in der Instanz abgelegt.

PersonID	Name	Vorname	Anschrift				.....
			Ort	PLZ	Straße	Land	

Es ist also in diesem Sinn nicht möglich, eine Anschrift wegzulassen. Sie kann höchstens leer sein, im Gegensatz zu Bestandteilen, die fehlen können. Außerdem können Angaben aus der Adresse nur im Zusammenhang einer PERSONEN-Instanz bereitgestellt werden.

Attributes als Schlüsselkomponenten

Alle Attributes können zur Bildung von Keys benutzt werden. Dabei ist es für Structured Attributes möglich, sowohl Properties als auch Keys der Attribute Structure als Schlüsselkomponenten zu referenzieren.

Beispiel Wenn beabsichtigt ist, PERSONEN später nach dem **Ort**, der in der **Anschrift** definiert ist zu sortieren, kann auf der Grundlage der vorangegangenen Beispiele ein entsprechender Schlüssel definiert werden:

```
structure Person {
   attribute (  ...
      Anschrift { type Adresse }                    )
   ....
   key sk_Ort component Anschrift.Ort;
}
```

Array Attributes

Attributes können als Mengen mit definierter Anzahl von Elementen, als Arrays (Bereiche) definiert werden. Arrays können in gleicher Weise für Literals als auch für Structured Attributes gebildet werden. Attribute Instanzen in Arrays werden direkt über ihre Position im Array (Index) angesprochen.

In der Structure-Instanz werden Arrays meistens in der vollen Länge angelegt. Das führt zu unnötigem Speicherverbrauch, wenn nicht alle Bereichselemente mit Werten belegt sind. In diesem Fall wird der Sachverhalt besser als Collection dargestellt, da in diesem Fall nur so viele Instanzen angelegt werden, wie tatsächlich benötigt werden.

Werden Attribute Arrays zur Schlüsselbildung herangezogen, können Multiple Keys gebildet werden. Dabei wird für jedes Array-Element ein Schlüsselwert gebildet.

3.2.3.2 References für Bestandteile

Teile eines Objektes oder einer Objektsicht werden durch References dargestellt. Die Grenze zwischen Attributes und References ist schwer zu ziehen. Der entscheidende Unterschied ist, daß Reference-Instanzen im Gegensatz zu Attribute-Instanzen nicht existieren müssen. References stellen Verweise auf einzelne Structure- oder auf Collections von Instanzen dar.

References in Structures sind Owning References, d.h. die Reference besitzt ihre Instanzen. Außerdem sind die Instanzen dieser Referenzen i.allg. nicht mehrfach referenziert, sondern nur über die Structure-Instanz erreichbar. Instanzen einer Reference müssen nicht unbedingt existieren, werden jedoch, wenn sie existieren, spätestens mit dem Löschen der übergeordneten Structure-Instanz gelöscht.

References können als Key Components verwendet werden. Da in diesem Fall die Key-Instanz nicht mehr von der Instanz selbst, sondern von der referenzierten Instanz abhängt (Wenn die Key-Instanz für eine PERSON aus dem **Namen** der **Mutter** gebildet wird, hängt sie nicht mehr von der PERSONEN-Instanz, sondern von der referenzierten **Mutter**-Instanz ab), wird der Key auch als indirekter Key bezeichnet. Indirekte Keys können auch als Multiple Keys definiert werden. Ein typisches Beispiel dafür sind Stichwortverzeichnisse, in denen mehrere Stichworte auf ein und den selben Sachverhalt verweisen. Singuläre References werden in Keys genauso wie Attributes behandelt. Sie unterscheiden sich nur hinsichtlich der Behandlung leerer References. Ist eine Referenz leer, wird kein Schlüsselwert erzeugt, während für Attributes immer ein Schlüsselwert im Index erzeugt wird.

Werden indirekte Keys über referenzierte Instanzen zur Bildung von Indizes verwendet, sichert ODABA2 die Konsistenz dieser Indizes, solange die Instanzen nicht mehrfach referenzierbar sind, also auch für einfache Collection References in Structures.

3.2.3.3 Relationships für Objektbeziehungen

Binäre Beziehungen zwischen unabhängigen Objekten werden aus einer bestimmten Sicht dargestellt. Relationships bilden die Beziehungen zwischen diesen Objektsichten ab, indem eine Beziehung zwischen den entsprechenden Structure-Instanzen hergestellt wird.

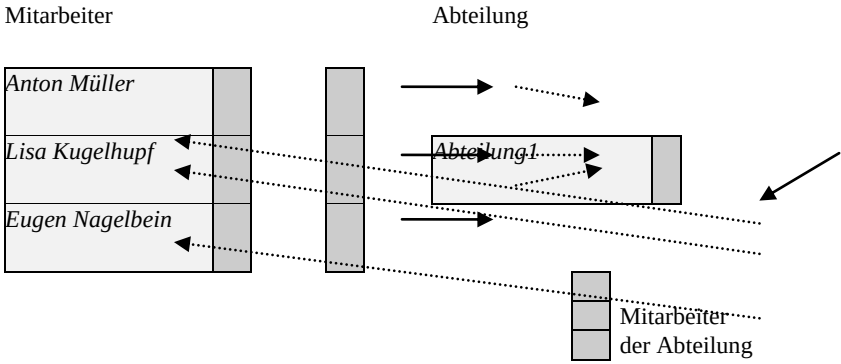
Relationships stellen eine spezielle Form funktionaler Zusammenhänge ohne eigene Qualität (ohne Merkmale auf der Ebene des Zusammenhangs) dar. Sie werden durch References in den beteiligten Structure-Instanzen dargestellt, die eine Beziehung miteinander eingehen.

Relationships sind in diesem Sinne spezielle References. Indem die References der ersten, der zweiten oder beider Structures als Reference Collection definiert werden, können über Relationships Mengenbeziehungen nicht nur als 1:1, sondern auch als 1:N oder M:N-Beziehungen dargestellt werden. Die References einer Relationship sind über ihre Möglichkeiten als Reference mit zusätzlichen Leistungen ausgestattet, die sich vor allem auf die Pflege der bidirektionalen Instanzenbeziehung beziehen.

Inverse References

Relationships werden in beiden Richtungen gleichberechtigt behandelt, d.h., wenn eine Instanz A eine Instanz B „kennt“, kennt B auch A. Es spielt nur eine sekundäre Rolle, ob A eine Beziehung zu B oder umgekehrt B eine Beziehung zu A aufnimmt. Dieser Zusammenhang wird durch inverse References dargestellt.

Beispiel Der MITARBEITER in einer Firma ist genau einer ABTEILUNG zugeordnet. Eine ABTEILUNG hat jedoch mehrere MITARBEITER, d.h. zwischen ABTEILUNG und MITARBEITER besteht eine 1:N-Beziehung. Diese wird auf der einen MITARBEITER-Seite durch eine singuläre, auf der ABTEILUNGS-Seite als Collection Reference dargestellt.



Die Instanz der ABTEILUNG ist als also von jedem MITARBEITER aus über jeweils eine singuläre Reference zu erreichen. Der Zugriff auf die MITARBEITER-Instanzen von der ABTEILUNG aus erfolgt über eine Collection Reference.

ODABA2 sichert die referentielle Integrität für Relationships sowohl beim einseitigen Herstellen als auch beim einseitigen Entfernen von Instanzen aus Relationship References.

Sekundäre Reference

Die symmetrische Darstellung von Relationships hat vor allem bei Kopiervorgängen den Nachteil, daß durch rekursives Kopieren erheblicher Aufwand entsteht. Aus diesem Grunde ist es oft sinnvoll, eine der References der Relation als sekundäre Reference zu definieren. Das hat beim Kopieren den Effekt, das nur Instanzen aus Primärreferenzen übernommen werden. Damit wird eine allzu tiefe Verschachtelung bei Kopiervorgängen verhindert.

Extent-basierte Relationships

Da Relationships Beziehungen zwischen freien Structure-Instanzen darstellen, sind die referenzierten Instanzen einer Relationship eine Teilmenge der freien Structure-Instanzen dieses Types, seines RootExtents. Oft ist es wünschenswert, Teilmenge von freien Structure-Instanzen zu definiert wird, die für die Bildung einer Beziehung zulässig sind. Dies kann über einen Basis-Extent geschehen, der die Menge der zugelassenen Instanzen einer Relationship festlegt.

Beispiel Wenn **Mitarbeitern** einer FIRMA ein FAHRZEUG zur Verfügung gestellt werden soll, wird eine Relationship zwischen FAHRZEUGEN und MITARBEITERN hergestellt. In diesem Fall kann es sinnvoll sein, sowohl die Menge der MITARBEITER zu begrenzen, die einen Anspruch auf ein FAHRZEUG haben, als auch die Menge der FAHRZEUGE, die MITARBEITERN zugeordnet werden können. Die Beziehung basiert also auf Teilmengen von MITARBEITERN und FAHRZEUGEN:

```
structure Fahrzeug { ...
   relationship
      Benutzer        { type Mitarbeiter; dimension 1;
                        based_on BerechtigteMA;
                        inverse Dienstfahrzeug        }
}

structure Mitarbeiter { ...
   relationship
      Dienstfahrzeug  { type Fahrzeug; dimension 1;
                        based_on Dienstfahrzeuge;
                        inverse Benutzer              }
}
```

Dienstfahrzeuge und **BerechtigteMA** sind dabei jeweils als eingeschränkte Extents für Fahrzeuge und Mitarbeiter definiert.

Da freie Structure-Instanzen vielfältige Beziehungen eingehen können, müssen diese Beziehungen kontrolliert werden, um z.B. beim Löschen einer Instanz alle vorhandenen inversen Beziehungen auflösen zu können. Deshalb werden Extent-basierte References für Relationships immer als Controlled References definiert.

Relationships können auch einseitig oder beidseitig als nicht Extent-basierte References definiert werden. In diesem Fall ist die Reference wie bei einfachen Structure References eine Owning Reference, deren Instanzen beim Löschen der referenzierenden Instanz gelöscht werden. Dabei werden auch die inversen References hergestellt.

Logische Objektbeziehung

Logische Objektbeziehungen beschreiben die Beziehung zwischen zwei Structure Instanzen über Key-Instanzen (Ident Key). Da beim Löschen einer Instanz die Beziehung aufgehoben wird, sind Relationships nur bedingt in der Lage, vergangene Zustände abzubilden. Aus diesem und auch aus Effizienzgründen ist es sinnvoll, Relationships einseitig zu definieren und auch einseitig aufrechtzuerhalten, wenn die referenzierte Instanz gelöscht wurde, so daß z.B. auch nach dem Konkurs einer Firma im Produkt sichtbar bleibt, wer ihr Hersteller war. Bei logischen References wird in diesem Sinne kein Zeiger auf die referenzierte Instanz, sondern wie in Relationen nur die Ident Key-Instanz in der Referenz gespeichert. Im Gegensatz zu Relationen können logische Objektbeziehungen jedoch auch multipel sein, d.h. es sind Beziehungen zu mehreren Instanzen möglich.

Abhängige Beziehungen

Da References für Relationships per Definition als Not Owning References definiert sind, gibt es keine existentiellen Abhängigkeiten zwischen der Instanz und den über Relationships referenzierten Instanzen. Zwischen Instanzen in einer Relationship können dennoch bestimmte Abhängigkeiten definiert werden.

Obwohl Abhängigkeitsbeziehungen einen kausalen Zusammenhang darstellen, werden sie von ODABA2 aus Kompatibilitätsgründen auch als Eigenschaften für

Relationships unterstützt. So kann die existentielle Abhängigkeiten von einer Relationships dahingehend definiert werden, daß eine Instanz nur solange existiert, wie sie Instanz dieser Relationship ist. Mit dem Entfernen abhängiger Instanzen aus der Relationship werden auch die Instanzen gelöscht. Wenn eine Instanz mehreren Relationships als **dependend** Instanz zugeordnet wurde, wird sie gelöscht, wen sie aus der ersten abhängigen Relationship gelöst wird. Als Folge des Löschens werden dabei auch alle abderen Beziehungen dieser Instanz gelöscht.

Beispiel Eine PERSON ist MITARBEITER nur im Zusammenhang mit einer FIRMA. Wird der Mitarbeiter entlassen oder löst sich die FIRMA auf, hört auch der Mitarbeiter in seiner Rolle als MITARBEITER (nicht als PERSON) auf, zu existieren, d.h. seine Mitarbeiter-Instanz muß entfernt werden. In diesem Sinne gibt es zwischen MITARBEITER und FIRMA einen existenziellen Zusammenhang.

```
structure Mitarbeiter {     ...
   relationship
      Anstellung  { type Firma; inverse Angestellte }
      ...               }
structure Firma {           ...
   relationship
      Angestellte { type Mitarbeiter; depends YES;
                    inverse Anstellung }
      ...         }
```

Die existentielle Abhängigkeit der MITARBEITER äußert sich darin, daß die MITARBEITER-Instanzen gelöscht werden, wenn die FIRMEN-Instanz aus der Relationship Reference entfernt wird.

Relationships als Key Components

References von Relationships können derzeit in ODABA2 nur als Key Components zur Bildung indirekter Keys verwendet werden, wenn sie nicht extent-basiert sind, wenn es sich also um Owning References handelt oder wenn eine inverse Reference definiert wurde. In diesem Fall gelten die gleichen Regeln wie für einfache References in Structures.

Der wesentliche Grund zur Definition indirekter Keys, nämlich die Bildung von Indizes für den sortierten Zugriff über diese, entfällt in vielen Fällen, da dieses Problem für extent-basierte References in Relationships durch die Ausnutzung inverser References günstiger gelöst werden kann.

Beispiel Wenn statt einer beliebigen Auswahl von **Stichworten** definierte DESKRIPTOREN zur Erstellung von Stichwortverzeichnissen benutzt werden sollen, können diese nicht mehr als References des BUCHES dargestellt werden, da die DESKRIPTOREN als freie Instanzen in einem Extent definiert sind und nicht mehr der Reference gehören. Statt dessen gibt es eine M:N-Beziehung zwischen BÜCHERN und DESKRIPTOREN, die als Relationship dargestellt wird.

```
structure Buch {
   attribute (
      Titel          { type STRING; size 100 }
      Autor          { type STRING; size 100 }
      ISBN           { type CHAR; size 10; }       )
   relationship
      Stichworte     { type Descriptor; dimension 10;
                       inverse Buecher;
                       order_by key_Stichwort     }
   key( key_Autor    component Autor;
        ident_key    component ISBN;              )
}
structure Descriptor {
   attribute
      Wort           { type CHAR; size 30 }
   relationship
      Buecher        { type Buch; inverse Stichworte;
                       order_by(key_Autor,ident_key)}
   key   key_Stichwort  component Wort;
}
```

Auf diese Weise ist ein stichwortbezogener Zugriff möglich. Außerdem können hier im Rahmen eines **Stichwortes** mehrere Sortierungen realisiert werden:

```
Stichwort().order_by(„key_Autor")
Stichwort().order_by(„ident_key")
```

Während im ersten Fall alle BÜCHER sortiert nach **Stichwort** und **Autoren** bereitgestellt werden, erscheinen sie im zweiten Fall sortiert nach **Stichwort** und **ISBN**.

3.2.3.4 BaseStructures für Basissichten

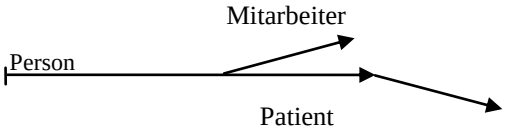
BaseStructures stellen Oberbegriffe oder abstrakte Sichten bezogen auf einen spezielleren Sachverhalt dar. In diesem Sinne bilden sie eine Beziehung (Relationship) zwischen einer speziellen und einer abstrakten Structure-Instanz ab. Die Besonderheit dieser Beziehung besteht darin, daß über BaseStructures eine

Beziehung zwischen zwei Instanzen, zwei Sichten zum gleichen Sachverhalt dargestellt wird, nämlich der Instanz der BaseStructure und der Instanz der abgeleiteten Structure.

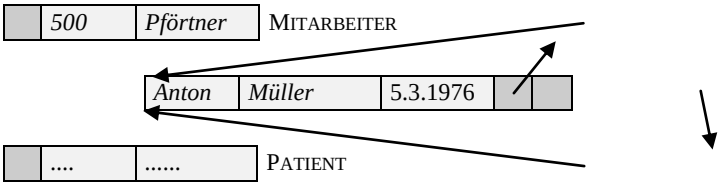
Mehrfachableitungen

Zwischen Basisinstanz und abgeleiteter Instanz kann eine 1:1- oder eine 1:N-Beziehung bestehen, d.h. es können von einer Instanz mehrere Instanzen abgeleitet werden. Wenn eine Structure-Instanz mehrere Ableitungen besitzt, wenn es also mehrere Structure-Instanzen gibt, die diese Instanz als Basisinstanz referenzieren, sprechen wir von Mehrfachableitung. Mehrfachableitungen können als spezielle Relationships dargestellt werden.

Beispiel MITARBEITER und PATIENTEN sind PERSONEN, können jedoch nicht als Spezialisierungen voneinander betrachtet werden, da nicht jeder PATIENT unbedingt ein MITARBEITER sein muß. Auch ist nicht jeder MITARBEITER ein PATIENT. Für die Darstellung einer PERSON als MITARBEITER und PATIENT ergibt sich dann folgende Instanzenhierarchie:



Indem die PERSONEN-Instanz in der MITARBEITER- und PATIENTEN-Instanz als Reference dargestellt wird, ist eine geteilte Benutzung der Basisinstanz möglich.



Das betrifft jedoch nur die Darstellung in der Datenbank. Beim Zugriff wird die Instanz der Base Structure immer entsprechend den Konventionen der Programmiersprache dargestellt (für C++ z.B. als direkt eingebettete Instanz).

Als Instanz einer Relationship können für die Basisinstanz die entsprechenden inverse References definiert werden, über die die Beziehungen zu den abgeleiteten Instanzen hergestellt werden.

Extent-basierte BaseStructures

In ODABA2 werden Extent-basierte BaseStructure-Instanzen über References dargestellt. In diesem Fall können auch inverse References definiert werden, so daß von einer BaseStructure-Instanz auch ihre abgeleiteten Instanzen erreichbar sind. Außerdem kann in diesem Fall die Kardinalität der Ableitungen begrenzt werden.

Instanzen für BaseStructures, die nicht auf einem Extent basieren, werden direkt in die abgeleitete Instanz eingebettet und verhalten sich wie Attribute-Instanzen in der übergeordneten Structure-Instanz.

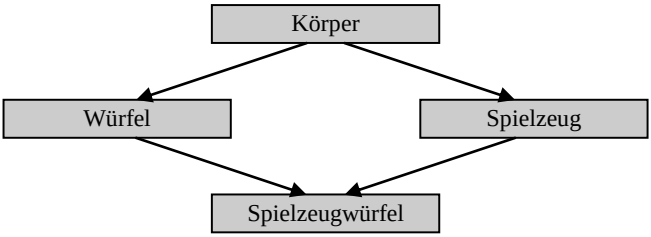
Mehrfachvererbung

Mehrfachvererbung bedeutet im Gegensatz zu Mehrfachableitung die Ableitung einer Structure von mehreren Base Structures. Mehrfachvererbung ist dann erforderlich, wenn mehrere unabhängige Sichten in einem Begriff vereint werden sollen.

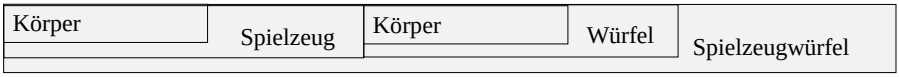
Beispiel Ein SPIELZEUGWÜRFEL ist sowohl SPIELZEUG als auch WÜRFEL. Da jedoch nicht jeder WÜRFEL ein SPIELZEUG, aber auch nicht jedes SPIELZEUG ein WÜRFEL ist, kann der SPIELZEUGWÜRFEL nur als Ableitung von WÜRFEL und SPIELZEUG dargestellt werden.

```
structure Spielzeugwuerfel {
     baseStruct ( wuerfel type Wuerfel;
                  spielzeug type Spielzeug )
}
structure Spielzeug {
     baseStruct   koerper type Koerper
}
structure Wuerfel {
     baseStruct   koerper type Koerper
}
structure Körper {
     attributes   volumen type INT
}
```

Sowohl SPIELZEUG als auch WÜRFEL sind ihrerseits Körper im geometrischen Sinn, d.h. es kann angenommen werden, daß sie jeweils von KÖRPER abgeleitet sind. Für den SPIELZEUGWÜRFEL ergäbe sich dann folgende Ableitungshierarchie:



Strukturell wäre KÖRPER in diesem Fall sowohl Bestandteil von WÜRFEL als auch von SPIELZEUG.



Um die auf diese Weise strukturell entstehende Redundanz auszuschließen, kann eine BaseStructure als **Virtual** BaseStructure definiert werden. Für Virtual BaseStructures wird in einer Ableitungshierarchie nur eine Instanz angelegt, auch wenn sie strukturell auf mehreren Wegen erzeugt wird. In der Datenbank können Virtual BaseStructures wieder als Extent-basierte Relationships über References dargestellt werden.

dominante Basisstruktur

Bei Bezug auf die Object Identity kann es zu Konflikten kommen, wenn bei Mehrfachableitung mehrere Basisinstanzen in einer abgeleiteten Instanz zusammengefaßt werden, die verschiedene Object Identities besitzen. In diesem Fall wird die Object Identity der dominanten BaseStructure von allen beteiligten Instanzen übernommen.

Beispiel Wenn eine bestehende SPIELZEUG-Instanz und eine WÜRFEL-Instanz, die bisher nicht als ein und dasselbe Objekt erkannt wurden und demzufolge mit verschiedenen Object Identities verbunden wurden, in einer Instanz zum SPIELZEUGWÜRFEL zusammengefaßt werden, erhalten nun alle drei Instanzen die Object Identity der dominanten BaseStructure-Instanz.

3.4 Real Objects für Subjekte

RealObjects stellen einen komplexen Sachverhalt dar, der durch eine Vielzahl von Zusammenhängen bestimmt ist. Ein reales Objekt ist genauso durch die Sichten bestimmt, die mit ihm verbunden sind, wie auch durch die inneren Objekte, d.h. durch die Objekte, die Bestandteile des RealObjects sind, und durch die Sichten auf andere Objekte.

RealObjects werden als Object-Instanzen dargestellt. Da auch eine Anwendung einen komplexen Sachverhalt beschreibt, wird auch einer Anwendung oder der mit ihr verbundenen Datenbank ein Objekt, das sogenannte RootObject, zugrunde liegen. Dieses RootObject stellt das Subjekt dar, welches den darzustellenden Sachverhalte begründet. Dabei kann es sich um eine Firma oder Einrichtung, um einen Produktionsprozeß oder um andere komplexe Sachverhalte handeln. Durch die Darstellung des Subjektes als Objekt ist es prinzipiell möglich, das Subjekt später in einen übergeordneten Kontext zu integrieren, ohne daß der individuelle Charakter des Subjektes verloren geht.

Sichten auf RealObjects

Auf ein RealObject kann es eine Vielzahl von Sichten geben, die nicht unbedingt über eine Hierarchie von Begriffen dargestellt werden können. Das Objekt als solches ist Träger aller dieser Sichten. Das bedeutet für die Darstellung von Objekt, daß sie zumindest einige ihrer Sichten kennen, da die Objekte auf diese durch ihr Verhalten aktiv Einfluß nehmen können.

> **Beispiel** Da eine FIRMA die Sicht des Finanzamtes kennt, nimmt sie aktiv Anteil daran, diese Sicht durch Eigendarstellung auf den aktuellen Stand zu bringen. Die FIRMA berichtet also über **Umsätze** und **Gewinne** oder **Verluste**, die als Folge ihres Verhaltens entstanden sind.

Die verschiedenartigen Sichten auf ein Objekt werden als Collection verschiedenartiger Structure-Instanzen im RealObject dargestellt. Darüber hinaus gibt es aber auch Sichten, die dem Objekt unbekannt bleiben, d.h. es werden nicht unbedingt alle Sichten auf ein Objekt mit diesem verbunden.

Sichten auf andere Objekte

Das Objekt kann als Subjekt auch selbst Sichten auf andere Objekte entwickeln. Gerade diese (An)Sichten sind es, die das Handeln des Subjektes prägen.

> **Beispiel** Um die Zusammenhänge in einer FIRMA darzustellen, müssen die vielfältigen Beziehungen zwischen ABTEILUNGEN, BEREICHEN und MITARBEITERN in verschiedenen Positionen dargestellt werden. Diese werden aus der Sicht, im Rahmen der Interessen der FIRMA in entsprechenden Structure-Instanzen abgebildet. In diesem Zusammenhang enthalten diese Instanzen genau die Informationen, die im Rahmen der FIRMA von Bedeutung sind. In anderen Zusammenhängen, in der Sicht anderer Subjekte, wird es sehr differenzierte Darstellungen der gleichen Objekte geben.

Die Sichten auf andere RealObjects werden als Structure-Instanzen dargestellt und in freien Collections oder Extents verwaltet. Somit besitzt jedes RealObject seine eigenen Extents. Ein spezieller Extent verwaltet die untergeordneten RealObjects.

Untergeordnete Objekte

RealObjects bestehen aus Teilen, die ihrerseits wieder RealObjects darstellen. Diese Teile sind oft von wesentlicher Bedeutung für den inneren Wirkungsmechanismus des Objektes. Untergeordnete RealObjects können als RealObject Instanzen dargestellt werden, die in einem Extent des übergeordneten Objekts zusammengefaßt werden. Dabei handelt es sich nicht um Sichten auf andere Objekte, sondern um Objekte, die dem übergeordneten Objekt gehören. Diese Zusammengehörigkeit ist meistens durch eine räumliche oder existentielle Abhängigkeit gegeben. Theoretisch ist es nahezu unmöglich, die Grenzen eines Objektes hinsichtlich seiner Teile genau zu bestimmen. Praktisch gestaltet sich die Zuordnung in den meisten Fällen jedoch problemlos.

Named Objects

Da es keine Bezeichnung für das Objekt an sich gibt, wird auf RealObjects i.allg. über eine Sicht Bezug genommen. Wenn wir aber auf ein RealObject Bezug nehmen wollen, ohne von einer speziellen Sicht auszugehen, muß das RealObject eine Bezeichnung, einen Namen erhalten, anhand dessen es identifiziert werden

kann. Indem RealObjects mit Namen verbunden werden, kann über diese Namen auf untergeordnete RealObjects Bezug genommen werden.

3.5 Pflege der Indizes und Beziehungen

Aus den vielfältigen Möglichkeiten, Zusammenhänge in der Datenbank darzustellen, ergeben sich einige Anforderungen an die Konsistentsicherung der Darstellung eben dieser Zusammenhänge. Die wesentlichen Probleme und die Art ihrer Lösung soll hier kurz umrissen werden. Durch ODABA2 werden folgende Zusammenhänge automatisch (online) gepflegt

- inverse References für Relationships
- Konsistenz der Extent-Hierarchien
- Konsistenz der Indizes für kontrollierte Collections
- Konsistenz der Beziehungen zwischen abgeleiteten und Basisinstanzen

Daraus resultieren einige spezielle Verhaltensweisen, die in diesem Abschnitt anhand einiger Beispiele kommentiert werden sollen. Sie können dieses Verhalten in der SAMPL-Datenbank einfach nachvollziehen. ODABA2 ist eines der wenigen OODBS, das die *formale logische Konsistenz* (Konsistenz der References und Indizes) der Datenbank sowohl im Single- als auch im Multi-User-Betrieb sichert.

3.5.1 Referenzen und Instanzen

In ODABA werden alle Mengen (Collections) durch Referenzen auf Instanzen dargestellt. Das Entfernen einer Instanz aus einer Collection bedeutet also i.allg. nur das Entfernen einer Instanzenreferenz, nicht der Instanz selbst. Andererseits gibt es aber zu jeder Instanz genau eine Collection, der die Instanz gehört. Dies sind die RootExtents, die References einer Structure und nicht Extent-basierte Relationships. Wenn eine Instanz aus einer solchen Owning Collection entfernt wird, wird nicht nur die Referenz auf die Instanz, sondern auch die Instanz selbst gelöscht. Das hat folgende Konsequenzen:

1. Hinzufügen und Löschen von Instanzen in RootExtents

Wenn Sie Instanzen zu RootExtents hinzufügen, wird eine neue Instanz erzeugt. Wenn Sie sich z.B. ein neues Thema *T1* zum Extent **Thema** (RootExtent)

hinzufügen, wird eine neue Themen-Instanz mit dem angegebenen Namen gebildet.

Wenn Sie das Thema *T1* wieder aus dem Extent **Thema** entfernen, wird die Instanz nicht nur aus dem Extent entfernt, sondern vollständig gelöscht, da der Extent **Thema** RootExtent ist und somit Besitzer seiner Instanzen.

2. Hinzufügen und Löschen in abgeleiteten Extents und Relationships

Wenn Sie hingegen eine Instanz zu einem abgeleiteten Extent oder einer Extent-basierten Relationship hinzufügen (z.B. über den Namen der Instanz, den IdentKey), wird zuerst geprüft, ob die Instanz bereits im BaseExtent oder im RootExtent referenziert wird. Wird sie nicht gefunden, wird eine neue Instanz erzeugt und dem RootExtent hinzugefügt. Wenn Sie also eine neue Stelle *TS* zum abgeleiteten Extent **TopStellen** hinzufügen, wird diese, da sie noch nicht existiert, im RootExtent **Stelle** neu angelegt und anschließend dem Extent **TopStellen** hinzugefügt.

Wenn Sie nun die Topstelle *TS* aus dem Extent **TopStellen** entfernen; wird nur die Referenz aus diesem Extent entfernt. Die Instanz selbst ist jedoch weiter über den Extent **Stelle** (alle Stellen) verfügbar.

3.5.2 Extent-Hierarchien

Extent-Hierarchien sind Hierarchien von Teilmengen. Abgeleitete Extents sind in diesem Sinne immer Teilmengen ihrer übergeordneten Extents (BaseExtents). Somit sind also offene Stellen (**StellenOffen**) und Topstellen (**TopStellen**) Teilmengen der Menge aller Stellen (**Stelle**). Die Menge der offenen Topstellen (**TopStellenOffen**) ist sowohl eine Teilmenge von **TopStellen** als auch von **StellenOffen** (und somit zwangsläufig eine Teilmenge des Durchschnitts von **TopStellen** und **StellenOffen**).

Durch diese Teilmengenbeziehungen sind bestimmte Abhängigkeiten zwischen den Extents definiert, die durch die Datenbank automatisch gepflegt werden:

1. Wird eine Instanz zu einem abgeleiteten Extent (Teilmenge) hinzugefügt, wird sie auch zu allen BaseExtents, zu den BaseExtents der BaseExtents usw. bis zum RootExtent, hinzugefügt. Mit dem Hinzufügen einer Stelle *TS* zu den **TopStellen** erscheint diese also auch im Extent **Stelle**. Wird die Stelle *TS* zum Extent **TopStellenOffen** hinzugefügt, wird sie sowohl zum Extent **TopStellen** als auch zum Extent **StellenOffen** hinzugefügt und in der Folge auch zum Extent **Stellen**.

2. Wird eine Instanz hinzugefügt, die nicht im RootExtent existiert, wird die Instanz im RootExtent erzeugt und anschließend allen Extents entlang der Hierarchie hinzugefügt (siehe 1.). Existiert die Instanz bereits im RootExtent, wird die existierende Instanz den entsprechenden Extents hinzugefügt.

3. Wird eine Instanz aus einem Extent entfernt, wird sie auch aus allen abgeleiteten Extents entfernt. Mit dem Löschen der Stelle *TS* aus dem Extent **StellenOffen** wird diese also auch aus dem abgeleiteten Extent **TopStellenOffen** entfernt. Sie ist jedoch nach wie vor im Extent **TopStellen** und im Extent **Stellen** enthalten.

3.5.3 spezielle Extent-Ableitungen

Die Bildung globaler Teilmengen (Extent-Ableitungen) kann mit speziellen Optionen versehen werden, die ein verändertes Verhalten der Extents zur Folge haben. So wurden im Beispiel SAMPL die abgeleiteten **Aufgaben**-Extents **AufgabenOffen**, **AufgabenBearb** und **AufgabenErled** als disjunkte abgeleitete Extents definiert, um zu verhindern, daß eine Aufgabe gleichzeitig in mehreren dieser Extents referenziert wird. Außerdem wurde festgelegt, daß der Extent **Aufgabe** die echte Vereinigungsmenge (union) seiner abgeleiteten Extents sein soll. Auf diese Weise wird gesichert, daß jede Aufgabe wenigstens in einem der abgeleiteten Extents enthalten ist. Die dritte Form spezieller Extent-Ableitungen finden Sie in der Extent-Hierarchie zu Stelle. Hier wurde der Extent **TopStellenOffen** als Durchschnitt seiner BaseExtents (intersect) definiert, wodurch gesichert ist, daß alle Stellen, die sowohl offene als auch Topstellen sind, auch Instanzen des Extents **TopStellenOffen** sind.

- **echte Vereinigungsmenge (union)**

  Wenn Sie mehrere Extents von einem BaseExtents ableiten (wie **AufgabenOffen**, **AufgabenBearb** und **AufgabenErled**), ist der BaseExtent per Definition eine Obermenge der Vereinigungsmenge seiner abgeleiteten Extents, d.h. er enthält mindestens alle Instanzen seiner abgeleiteten Extents. In diesem Fall soll er jedoch genau die Instanzen seiner abgeleiteten Extents enthalten, d.h. jede Aufgabe soll in mindestens einem der abgeleiteten Extents enthalten sein. Um diesen Zusammenhang sicherzustellen, reagieren BaseExtents mit der union-Option in folgender Weise auf Veränderungen am Extent:

  1. Wird eine Instanz zum BaseExtent (Vereinigungsmenge) hinzugefügt, wird sie auch zum ersten der abgeleiteten Extents hinzugefügt. In diesem Sinne ist der erste abgeleitete Extent ein ausgezeichneter Extent, der als Standard-Extents angesehen wird. Da im Beispiel der erste der abgeleiteten Extents **AufgabenBearb** ist, wird eine Aufgabe *A1*, die über den Extent **Aufgabe** hinzugefügt wird, auch dem Extent **AufgabenBearb** (Menge der bearbeiteten Aufgaben) hinzugefügt.

  2. Wird eine Instanz aus einem der abgeleiteten Extents entfernt, so wird sie auch aus dem BaseExtent entfernt, wenn sie nicht in einem der anderen abgeleiteten Extents enthalten ist. Wenn also die Aufgabe *A1* aus dem Extent **AufgabenBearb** entfernt wird, wird sie auch aus dem Extent **Aufgabe** entfernt (wenn sie nicht noch in einem anderen der abgeleiteten Extents enthalten ist). Da der Extent **Aufgabe** der RootExtent ist, dem die Instanzen gehören, wird die Aufgabe *A1* in diesem Fall auch als Instanz gelöscht.

- **disjunkte Extent-Ableitungen**

  Wenn Sie mehrere Extents von einem BaseExtents ableiten (wie **AufgabenOffen**, **AufgabenBearb** und **AufgabenErled**), können Sie durch Angabe der disjunkt-Option sichern, daß eine Instanz in höchstens einem der abgeleiteten Extents referenziert werden kann. Auch daraus ergeben sich spezielle Verhaltensweisen der Extents:

1. Wenn Sie eine Instanz zu einem abgeleiteten Extent hinzufügen, der zu anderen abgeleiteten Extents disjunkt sein soll, wird die Instanz, falls sie in einem der anderen abgeleiteten Extents referenziert wird, aus diesem entfernt. Auf diese Weise können Sie die Aufgabe *A1* z.B. von den bearbeiteten Aufgaben (**AufgabenBearb**) in die Menge der erledigten Aufgaben verschieben, indem Sie *A1* zum Extent **AufgabenErled** hinzufügen. Aufgrund der disjunkt-Option wird *A1* dabei automatisch aus dem Extent **AufgabenBearb** entfernt, so daß die Bedingung auch nach dem Hinzufügen wieder erfüllt ist.

- **echter Durchschnitt (intersect)**

  Wenn Sie einen Extent von mehreren BaseExtents ableiten (wie **TopStellenOffen**), ist dieser per Definition eine Teilmenge des Durchschnitts all seiner BaseExtents, d.h. er enthält nur Instanzen, die in jedem der BaseExtents enthalten sind. Die offenen Topstellen sollen jedoch alle Stellen enthalten (echter Durchschnitt), die offene und Topstellen sind. Aus diesem Grund wurde der Extent **TopStellenOffen** im Beispiel als Durchschnitt seiner BaseExtents definiert. Das ist mit folgenden Konsequenzen verbunden:

  1. Wenn zu einem der BaseExtents eine Instanz hinzugefügt wird, wird geprüft, ob diese bereits in allen anderen BaseExtents enthalten ist. Ist dies der Fall, wird die Instanz auch dem Durchschnitt hinzugefügt. Wenn Sie also eine Stelle *TSO* zum Extent **TopStellen** und zum Extent **StellenOffen** hinzufügen, wird die Stelle *TSO* auch zum Extent **TopStellenOffen** hinzugefügt.

  2. Aus einem als Durchschnitt seiner BaseExtents definierten Extent können keine Instanzen entfernt werden. Sie können also die eben gebildete offene Topstelle *TSO* nicht direkt aus dem Extent **TopStellenOffen** entfernen, da dann die Bedingung für den echten Durchschnitt nicht mehr erfüllt wäre, ohne diese wenigstens aus einem der BaseExtents zu entfernen. Andererseits ist es nicht entscheidbar, aus welchem der BaseExtents die Instanz entfernt werden sollte, um die Konsistenz wieder herzustellen. Die Stelle *TSO* wird jedoch automatisch aus dem Extent **TopStellenOffen** entfernt, wenn sie aus dem Extent **StellenOffen** oder aus dem Extent **TopStellen** entfernt wird.

3.5.4 Relationships

Für Relationships ergeben sich weitere Besonderheiten, die zum einen auf der Tatsache beruhen, daß Relationships einen bidirektionalen Zusammenhang darstellen, zum anderen aber auch Collections sein können, die von Extents abgeleitet sind.

3.5.4.1 Inverse References

Da Relationships in ODABA über zwei References dargestellt werden, auf die jeweils separat zugegriffen werden kann, muß die Abhängigkeit dieser Beziehung gepflegt werden. Das wird in ODABA nur gesichert, wenn die References der Relationship jeweils unter Angabe der inversen Reference definiert wurden. In diesem Fall werden folgende Konsistenzbedingungen durch die Datenbank gesichert: Im Beispiel SAMPL wurde die Relationship **aufgaben-stelle** (Aufgaben einer Stelle bzw. die einer Aufgabe zugeordnete Stelle) als Relationship mit inversen References definiert.

1. Wenn eine Instanz zur Reference einer Relationship hinzugefügt wird, wird die Instanz, der diese Reference gehört, auch zur inversen Reference hinzugefügt. Wenn also der AUFGABE *A1* (Reference **stelle**) die STELLE *TSO* zugeordnet wird, wird die AUFGABE *A1* gleichzeitig auch der Menge der Aufgaben zugeordnet.

2. Wenn eine Instanz aus der Reference einer Relationship entfernt wird, wird die Instanz, der diese Reference gehört, auch aus der inversen Reference entfernt. Wenn also die AUFGABE *A1* aus der Menge der Aufgaben (Reference **aufgaben** in STELLE) entfernt wird, wird gleichzeitig auch die Zuordnung der STELLE TSO in der AUFGABE (Reference **stelle**) entfernt.

3.5.4.3 Extent-basierte Relationships

So wie die BaseStructures können auch Relationships über Extents definiert werden. So ist im Beispiel SAMPL die Beziehung zwischen Stellen und Aufgaben

(**stelle-aufgaben**) als Extent-basierte Relationship definiert worden, die für **stelle** auf dem Extent **Stelle** und für **aufgaben** auf dem Extent **AufgabeBearb** basiert.

Extent-basierte Relationships sind als Collections wie abgeleitete Extents von ihrem BaseExtent abgeleitet (Teilmengen des BaseExtent) und verhalten sich auch genauso. Fügen Sie also eine Aufgabe *A2* als AUFGABE der STELLE *TSO* hinzu, finden Sie *A2* anschließend auch in dem Extent **AufgabenBearb**, da **aufgaben in** STELLE auf dem Extent **AufgabenBearb** basiert.

Beispiel Wie komplex die Folgen sein können, die sich aus einer Kombination der bisher genannten Konsistenzregeln ergeben, soll dieses Beispiel verdeutlichen. Nachdem Sie in der Menge der offenen Aufgaben (über den Extent **AufgabenOffen**) eine AUFGABE *A1* hinzugefügt haben, können Sie in dieser nun über die Reference **stelle** eine Stelle (z.B. *TSO*) zuordnen, die die Bearbeitung der Aufgabe übernehmen soll. Wenn Sie die STELLE *TSO* zuordnen, werden folgende Maßnahmen zur Konsistenzsicherung durchgeführt:

1. Die AUFGABE *A1* wird in die **aufgaben** der STELLE *TSO* eingetragen (inverse Reference).
2. Da **aufgaben** in STELLE auf dem Extent **AufgabenBearb** basiert, wird die AUFGABE *A1* dem Extent **AufgabenBearb** hinzugefügt.
3. Da **AufgabenBearb** und **AufgabenOffen** disjunkt zueinander sind, wird *A1* aus dem Extent **AufgabenOffen** entfernt.

Nach dem Zuordnen der Stelle *TSO* ist *A1* also in der Menge der offenen Aufgaben, über die sie gerade bearbeitet wurde, nicht mehr enthalten. Statt dessen finden Sie *A1* in der Menge der bearbeiteten Aufgaben (**AufgabenBearb**) wieder, was eigentlich auch dem Sachverhalt entspricht, da die AUFGABE *A1* nun durch die STELLE *TSO* bearbeitet wird.

3.5.5 Extent-basierte Ableitungen

Extent-basierte Ableitungen ermöglichen es, daß Sie abgeleitete Instanzen für eigenständigen Basisinstanzen bilden können. Dies ist z.B. die Grundlage für mögliche Mehrfachableitungen. Im Beispiel SAMPL sind STELLE und AUFGABE Extent-basierte Ableitungen von THEMA. Wenn Sie also eine Stelle *S1* erzeugen (z.B. durch Hinzufügen zum RootExtent **Stelle**), wird nicht nur eine Stellen-Instanz, sondern auch eine THEMA-Instanz für die Basisklasse über den Extent **Thema** gebildet.

Die Basisinstanz zum Thema, ist jetzt sowohl über die Stellen (Extent **Stelle**), als auch über das Thema *S1* (Extent **Thema**) sichtbar. Wenn für die abgeleiteten Instanzen eine inverse Reference in der Basisinstanz definiert wurde, werden diese Zusammenhänge auch beim Löschen beachtet.

1. Mit dem Erzeugen abgeleiteter Instanzen Extent-basierter Ableitungen werden automatisch auch die Basisinstanzen als eigenständige Instanzen erzeugt. Löschen Sie hingegen die abgeleitete Instanz (also z.B. die Stelle *S1* über den RootExtent **Stelle**), wird zwar die STELLE-Instanz gelöscht, die Basisinstanz - das THEMA *S1* - bleibt hingegen als eigenständige Instanz erhalten.

2. Wenn Sie abgeleitete Instanzen erzeugen, für die bereits eine Basisinstanz existiert, wird diese zugewiesen. Wenn es bereits ein THEMA *S1* gibt, wenn die STELLE *S1* gebildet wird, wird die THEMA-Instanz *S1* als Basisinstanz der STELLE-Instanz *S1* zugeordnet. Die Zuordnung erfolgt anhand des Identschlüsselwertes.

3. Wenn Sie eine Instanz löschen, zu der es abgeleitete Instanzen gibt (Basisinstanz), werden auch die abgeleiteten Instanzen gelöscht, soweit inverse References definiert wurden. Löschen Sie also die THEMA-Instanz S1, wird auch die Stelle S1 gelöscht, die über die inverse Reference **planung** in THEMA referenziert wird.

3.5.7 Indizes von Collections

In ODABA werden Indizes von Collections i.allg. automatisch (online) aktualisiert. Das betrifft sowohl die Indizes für lokale Collections (Reference Collections) als auch Indizes für Extents. Wird also der Inhalt einer Instanz geändert, werden die Indizes aller Instanzen, die diese Instanz referenzieren, aktualisiert.

1. Mit der Änderung an Basisinstanzen sind ggf. Pflegemaßnahmen an den abgeleiteten Instanzen oder in Collections, die auf diese Instanzen verweisen, erforderlich. So müssen z.B. Titel-Indizes in den Extents **Stelle** und **Aufgabe** aktualisiert werden, wenn der Titel eines Themas geändert wird. Außerdem müssen abgeleitete Instanzen gelöscht werden, wenn ihre Basisinstanz gelöscht

wird. ODABA2 sichert diese Konsistenzbedingungen, wenn in der Basisinstanz eine inverse Reference (Relationship) auf die abgeleiteten Instanzen definiert wurde. Für Thema ist dies die Relationship **planung**. Die Pflege dieser Relationship wird vollständig durch das OODBS übernommen. Wenn in der BaseStructure inverse Relationships zu den abgeleiteten Structures hergestellt wurden sichert ODABA2, daß

- abgeleitete Instanzen entfernt werden, wenn eine ihrer Basisinstanzen gelöscht wird (wenn sie also das Thema GF4 löschen, wird auch die Stelle GF4 gelöscht)
- die Indizes der Collections für die abgeleiteten Instanzen gepflegt werden, wenn sich die Schlüsselwerte ändern (wenn Sie z.B. den Namen des Themas GF3 durch Umbenennen nach gf3 ändern, wird diese Änderung auch im Index für den Extent Stelle nachvollzogen).

3.6 Versionsbildung

Die Versionsbildung in ODABA2 erfolgt durch die Bildung von Instanzen- und Object-Versionen. Dadurch können sowohl Änderungen des Sachverhaltes als auch Änderungen der Sicht auf einen Sachverhalt ausgedrückt werden.

Versionen werden durch eine Versionsnummer ausgedrückt, die auf der Instanzen- und/oder auf der Objektebene abgelegt wird. Die Numerierung erfolgt durch ODABA2. Die Bildung von Versionen beinhaltet keine Angaben zu Datum und Uhrzeit. Sie stellen also nur in dem Sinne eine zeitliche Ordnung dar, als daß sie jüngere und ältere Instanzen kennen. Auf der RealObject-Ebene kann jedoch eine Versionstabelle erzeugt werden, über die jede Versionsnummer einem Zeitpunkt (DateTime) zugeordnet werden kann. Diese Zuordnung wird automatisch bei der Bildung neuer Object-Versionen vorgenommen.

ODABA2 unterstützt die Versionsbildung auf der Datenebene genauso, wie die Versionsbildung auf der Schemaebene. Dadurch wird es möglich, auch zur Laufzeit von Projekten Schemaänderungen durchzuführen, ohne daß umfassende Reorganisationsmaßnahmen erforderlich sind.

Da Versionen einer Instanz integraler Bestandteil der Instanz sind, stellt sich die Frage, ob Versionen einer Instanz beim Kopieren mit übernommen werden oder nicht. ODABA2 läßt hier beide Varianten zu, d.h. Instanzen können mit oder ohne ihre Geschichte kopiert werden. Dadurch wird es möglich, den Umfang einer Datenbank zu reduzieren, indem sie ohne Übernahme der Geschichte kopiert und somit auf die aktuellen Instanzen reduziert wird.

3.6.1 Versionsbildung für Instanzen

Die Versionsbildung der Daten auf der Instanzenebene erfolgt beim Speichern der Instanz. Zu diesem Zeitpunkt kann entschieden werden, ob die aktuelle Version einer Instanz überschrieben werden soll, oder ob eine neue Version der Instanz gebildet werden soll.

Beispiel Wenn sich der Name einer Person ändert, kann es sinnvoll sein, sich den alten Namen zu merken. Das könnte durch Erzeugen einer neuen Instanzenversion geschehen.

```
PI(Person) pers_pi(&dbh,"Mitarbeiter",PI_Write);
DBField       name(pers_pi,"Name");
pers_pi.Get("Müller|Anton");
Name = "Schulz";      // Änderung Personen-Instanz
pers_pi.NewVersion(); // Erzeugen Instanzenversion
```

Die Versionsnummern werden dabei automatisch gebildet. Das Lesen einer älteren Version kann ausgehend von der aktuellen Instanz erfolgen:

```
PI(Person)  pers_pi(&dbh,"Mitarbeiter",PI_Write);
pers_pi.Get("Müller|Anton");
pers_pi.PreviousVersion(); // Vorgängerversion
```

Ebenso ist es möglich, durch Angabe der Versionsnummer (Object-Version, Instanzenversion) gezielt auf eine bestimmte Version zuzugreifen oder ausgehend von der eingestellten Version eine bestimmte Anzahl von Schritten zurückzugehen.

Die Versionsnummern werden i.allg. nicht in der Instanz abgelegt, so daß aus der aktuellen Instanz nicht hervorgeht, welches die aktuelle Version der Instanz ist. Wenn jedoch in der Structure ein Versions-Attribute mit dem Namen **__VERSION** definiert wurde, wird die aktuelle Versionsnummer in diesem Feld beim Bilden einer neuen Version automatisch eingetragen. Beim Anlegen einer Instanz hat diese

immer die Versionsnummer 0. Mit jedem Bilden einer neuen Version wird die Versionsnummer um eins erhöht.

Hinweis Da die Versionsbildung auf der Entry-Ebene angesiedelt ist, also auf der Ebene der physikalischen Datenbankinstanzen, zu denen auch Indizes oder Teile mehrstufiger Indizes gehören, können prinzipiell alle Datenbankinstanzen versioniert werden, also Indizes ebenso, wie freie Structure Instanzen. In diesem Fall werden jedoch nur Funktionen zur Bildung von Versionen auf der Ebene der Structure-Instanzen bereitgestellt. Structure-Instanzen werden hinsichtlich ihrer Attribute und ihrer Base Structures versioniert, nicht aber bezüglich ihrer Reference Collections.

Das führt dazu, daß ältere Instanzenversionen i.allg. nicht den Konsistenzanforderungen der Datenbank entsprechen. Sowohl die Indizes als auch die References älterer Versionen werden bei der Bildung von Instanzenversionen nicht gepflegt. In diesem Sinne können ältere Instanzenversionen zwar Hinweise darauf geben, welchen Zustand eine Instanz zu einem zurückliegenden Zeitpunkt hatte. References und Beziehungen widerspiegeln jedoch nicht in jedem Fall den damaligen Stand. Dieser Mangel kann durch die Bildung von Object-Versionen behoben werden, die Sachverhalte in ihrer gesamten Komplexität versionieren.

Ältere Instanzenversionen können beschränkt geändert, nicht aber erneut versioniert werden. Bei der Änderung an älteren Instanzen-Versionen werden keine konsistenzerhaltenden Maßnahmen wie Indexpflege oder Gültigkeitsprüfungen ausgelöst.

3.6.2 Versionsbildung für RealObjects

Die Versionsbildung für RealObjects erfolgt, indem eine neue Version für ein RealObject erzeugt wird. In diesem Fall werden alle Instanzen des RealObjects, die nach dem Erzeugen einer neuen Version geändert werden, automatisch versioniert. Das schließt nicht nur nachgeordnete Instanzen, sondern auch alle Indexinstanzen ein. Auf diese Weise wird bei der Versionsbildung für RealObjects die Konsistenz innerhalb jeder Object-Version gesichert.

Beispiel Wenn eine neue Object-Version erzeugt werden soll, wird dies einfach dem entsprechenden RealObject mitgeteilt. Erst bei der Speicherung von geänderten Instanzen werden jedoch tatsächlich neue Versionen für die geänderten Instanzen erzeugt.

```
ACObject      sample(&dbhandle,„Sample“,PI_Write);
sample.NewVersion(); // Erzeugen neue Object-Version
```

Die Versionsnummer wird dabei automatisch gebildet und dem aktuellen Tagesdatum und der Uhrzeit zugeordnet (Datum und Uhrzeit können auch als Parameter übergeben werden). Das Einstellen auf eine ältere Object-Version kann dann wieder ausgehend von der aktuellen Instanz erfolgen.

```
ACObject      sample(&dbhandle,„Sample",PI_Write);
sample.PreviousVersion();          //          Einstellen
Vorgängerversion
```

Durch Angebe eines Zeitpunktes (DateTime) oder einer Versionsnummer kann das RealObject auch auf weiter zurückliegende Object-Versionen eingestellt werden.

Wenn die Versionsbildung auf der Object-Ebene erfolgt, ist auf der Instanzenebene keine explizite Versonsbildung möglich. Die Versionsbildung erfolgt automatisch, wenn eine Instanz geändert wird. Auch in diesem Fall kann in den Structures eine Versionsnummer (__VERSION) definiert werden, in dem die aktuelle Versionsnummer gespeichert wird. Dies ist hier jedoch immer die Nummer der Object-Version, zu der die letzte Änderung der Instanz erfolgte.

Ebenso, wie bei der Bildung von Instanzenversionen können auch hier ältere Versionen einer Instanz bereitgestellt werden. Wenn jedoch auch über ältere Indexversionen zugegriffen werden soll, muß das RealObject auf die entsprechende Version eingestellt werden.

Beispiel Wenn eine ältere Object-Version bearbeitet werden soll, muß das RealObject auf die entsprechende Version eingestellt werden.

```
ACObject      sample(&dbhandle,„Sample",PI_Write);
sample.PrevVersion(); // vorhergehende Object-Version
```

Durch Angabe eines Zeitpunktes (Date, DateTime) oder einer Versionsnummer kann auch auf eine weiter zurückliegende Version eingestellt werden.

Für RealObjects kann ebenso wie für Instanzen die Bearbeitung älterer Object-Versionen erfolgen. Dabei wird auch für ältere Object-Versionen ein konsistenter Zustand bezüglich der Indizes und der definierten Konsistenzregeln gesichert. Da es dabei allerdings zu Konflikten mit der neuen Object-Version kommen kann, sind Änderungen in der alten Version nur solange durchführbar, wie keine Instanzen davon betroffen sind, die bereits in der neuen Object-Version existieren. Sowie also eine neue Version der Instanz in der neuen Object-Version erzeugt wurde, sind

Änderungen an dieser Instanz in der alten Object-Version nur noch möglich, solange keine zeitlichen Konsistenzbedingungen verletzt werden.

Beispiel Sie können in einer alteren Version keine Instanz löschen, die in einer neueren Version existiert, da eine der zeitlichen Konsistenzbedingungen darin besteht, daß gelöschte Instanzen nicht wiederauferstehen können.

Die Anzahl der Objektversionen ist derzeit ebenso wie die Anzahl der Instanzenversionen auf 16 Millionen begrenzt.

3.6.3 Versionsbildung auf der Schemaebene

Die Versionsbildung auf der Schemaebene erfolgt als Objektversion, da durch die vielfältigen Zusammenhänge zwischen den Instanzen der Schemaebene die Versionierung einzelner Instanzen nicht sinnvoll ist. Objekte auf der Schemaebene sind Projekte oder Sub-Projekte (Pakete). Mit der Bildung einer neuen Version auf der Schemaebene wird also eine neue Version für das gesamte Schema oder Repository für ein Projekt oder ein Paket gebildet.

Da jedes Projekt/Paket in einem Repository in einem eigenständigen RealObject dargestellt wird, kann für jedes Projekt/Paket separat die Bildung von Versionen erfolgen. Mit der Versionierung werden sowohl die Structure-Definitionen als auch die Methoden und kausalen Zusammenhänge versioniert.

Nach der Bildung einer neuen Projektversion auf der Schemaebene können Structures, die als „ready“ markiert waren, wieder geändert werden. Diese Änderungen werden für die Speicherung der Instanzen jedoch erst dann wirksam, wenn die Projektversion als „ready“ definiert wurde. Von der Bildung einer neuen Projektversion bis zur Markierung der Projektversion als „ready“ werden die Structure-Instanzen in der Datenbank mit der Vorgängerversion (also entsprechend dem alten Definitionsstand) gelesen und geschrieben. Das gilt auch für die definierten Aktionen und Expressions.

Nach dem Ready-setzen der neuen Projektversion erfolgt die dynamisch Anpassung der Instanzen in der Datenbank an die neuen Structure-Definitionen. Beim Einlesen werden Instanzen älterer Structure-Versionen dem aktuellen Stand angepaßt (Versionskonvertierung). Bei Änderungen werden sie dann zum jeweils aktuellen

Strukturstand gespeichert. Diese Versionskonvertierung löst ein VConvertion-Event aus, so daß eine applikationsbezogene Behandlung bei der Versionskonvertierung möglich ist.

Die Versionskonvertierung wird jedoch nur von älteren Versionen zu neueren hin ausgeführt, d.h. es ist nicht möglich, Instanzen neuerer Structure-Versionen in ältere zu überführen. In diesem Sinne wird nur die Aufwärtskompatibilität gewährleistet. Durch Einstellen einer älteren Version beim Starten einer Applikation ist es zwar möglich, mit einer älteren Projektversion zu arbeiten. Instanzen können jedoch nur gelesen oder geändert werden, solange sie noch nicht mit einer neueren Structure-Version gespeichert wurden. Wurden für die Objekte Objektversionen gebildet, wird das RealObject beim Öffnen automatisch auf die höchste zulässige Version für die Projektversion eingestellt.

Die Anzahl der möglichen Projektversionen ist auf 254 begrenzt. Da Projektversionen etwa alle zwei bis drei Monate gebildet werden, entspricht dies etwa einem Zeitraum von 50 Jahren. Durch die Reorganisation der Datenbank kann die Projektversion wieder auf 0 gesetzt werden (das ist u.U. auch ratsam, um die erforderlichen Instanzenkonvertierungen zu vermeiden).

Kapitel 4
Das funktionale Modell

So, wie die Merkmale der Sachverhalte im Objektmodell in Structure-Instanzen abgebildet werden, wird das Verhalten im funktionalen Modell auf **Methoden** abgebildet. Dabei wird das Verhalten nicht auf das konkrete Objekt bezogen dargestellt, sondern als arttypisches Verhalten auf der Schemaebene beschrieben. Methoden bilden also potentielles, mögliches Verhalten ab. Das Verhalten wird wie die strukturellen Eigenschaften eines Objektes im Rahmen einer Sicht, also einer Structure dargestellt. Die Erweiterung einer Structure um die Spezifikation der Methoden wird als Class bezeichnet. Die Class ist also die Zusammenfassung von Structure und Methoden. In ODABA2 können Methoden auf der Ebene von Implementierungsklassen definiert werden, d.h., eine Structure muß zur Implementierungsklasse spezialisiert werden, wenn Methoden für sie spezifiziert werden sollen.

Neben der Möglichkeit, Methoden als Programmfunktion zu implementieren, werden in ODABA2 verschiedene Implementierungstechniken unterstützt, so daß auch ein Window oder ein Dokument als Methode dargestellt werden kann. Dennoch ist die häufigste Form die Implementierung als Programm in einer objektorientierten Programmiersprache. Die ODABA2-Entwicklungsumgebung unterstützt gegenwärtig vor allem C++, kann aber auch für andere Programmiersprachen aufbereitet werden.

Das funktionale Modell von ODABA2 ist eine strukturelle und methodische Erweiterung des ODABA2-Objektmodells. Durch Implementierung neuer Begriffe auf der ODABA2-Modellebene entstand das funktionale ODABA2-Modell, in dem nicht nur Structures definiert, sondern auch Classes erzeugt und Methoden implementiert werden können. Damit ist das funktionale Modell von ODABA2 eng mit der ODABA2-Entwicklungsumgebung verbunden. Da ODABA2 aber nicht nur ein rekursives, sondern auch ein völlig offenes Modell ist, kann sowohl das Objektmodell zu einem beliebigen anderen funktionalen Modell erweitert werden,

als auch das ODABA2-Modell unter Beibehaltung seiner Funktionalität erweitert werden.

Das ODABA2-Modell umfaßt verschiedene Ebenen, auf denen Funktionalität bereitgestellt werden kann. Je nach Art und Ebene der Implementierung gibt es spezielle Class-Ebenen und implementierungsabhängige Arten von Classes. Die verschiedenen Ebenen für die Implementierung von Verhalten ergeben sich aus verschiedenen Abstraktionsniveaus einer Sicht und finden in den verschiedenen Arten von Implementierungsklassen ihren Ausdruck.

- **System Classes**

  System Classes sind Classes, die von ODABA2 zur Verwaltung der persistenten Instanzen bereitgestellt werden. Über die Methoden der System Classes werden Funktionen wie Hinzufügen, Lesen oder Löschen von Instanzen realisiert.

- **Problem Classes**

  Methoden auf der Ebene der Problem Structures entsprechen dem typischen Verhalten der problemrelevanten Objekte aus bestimmten Sichten. Problem Classes werden durch den Applikationsentwickler implementiert.

- **Context Classes**

  Context Classes beschreiben das Verhalten in bestimmten Zusammenhängen. Oft entsteht im Rahmen einer Rollenspezialisierung auch ein spezialisiertes Verhalten, das im Rahmen von Context Classes dargestellt wird.

4.1 System Classes

System Classes stellen die Funktionalität bereit, die erforderlich ist, um im Rahmen eines OODBS auf persistente Instanzen zuzugreifen. In ODABA2 werden System Classes können für folgende Structures der Modellebene bereitgestellt:

- **Dictionary - Datenbank-Handle für Ressourcendatenbanken**

  Dictionary-Funktionen dienen der Bereitstelung von Informationen zur Structure-Definition, sowie zu Extent- und Indexdefinitonen.

- **DBHandle - Datenbank-Handle**

  Datenbank-Handle können für Datenbanken beliebiger Ebenen erzeugt werden. Auch das Dictionary ist ein Datenbank-Handle. Datenbank-Handle müssen erzeugt werden, um Zugriff auf eine Datenbank zu erlangen.

- **ACObject - Objekt-Handle**

  Über Objekt-Handle wird der Zugang zu einem Real Object möglich. Da alle Instanzen über Extents Real Objects zugeordnet sind, ist ein Object-Handle erforderlich, um auf Instanzen zuzugreifen. Da aber jeder Datenbank ein Real Object, das Root Object, repräsentiert, ist das Datenbank-Handle auch ein Objekt-Handle.

- **PI-Handle - Handle/Iterator zum Zugriff auf persistente Instanzen**

  PI-Handle ermöglichen auf Objekt-Ebene den Zugriff zu den Instanzen der Extents. Reference Properties in den Instanzen werden intern wiederum als PI-Handle dargestellt, so daß über Reference Properties mittels PI-Funktionen der Zugriff auf referenzierte Instanzen realisiert wird.

- **Property - Property-Handle**

  Property Handle enthalten die Funktionalität zum Zugriff auf die Property-Instanzen. Das schließt sowohl verschiedene Konvertierungs-Funktionen als auch kontrollierte Änderungen mit ein.

System Classes verbinden die Instanzen, also die Daten der jeweiligen Ebene, mit ihrer Schemadefinition. Jede System Class kennt also die Schemadefinition des entsprechenden Sachverhaltes. Somit sind System Classes in der Lage, sehr flexibel auf Änderungen der Schemaebene zu reagieren. Mit Hilfe der System Classes ergeben sich spezielle Implementierungstechniken, die die Methoden weitgehend unempfindlich gegen Änderungen der Schemaebene, z.B. Änderungen der Structure-Definition, machen.

Im folgenden werden die System Classes grob charakterisiert. Eine detaillierte Beschreibung ihrer Leistung kann dem Referenz-Handbuch (Datenbankklassen) entnommen werden, das als Online-Hilfe vorliegt.

4.1.1 System Class Dictionary

ODABA2 geht davon aus, daß die Structures der Modellebene in eine interne Darstellung konvertiert werden, um die Zugriffseffizienz zu verbessern. Das Dictionary ist in diesem Sinne ein Real Object, das das Objektmodell einer Anwendung repäsentiert. Diese Repräsentation erfolgt in gleicher Weise im Internen als auch im Datenbankformat. Auf die wichtigsten Structures der Modellebene kann also sowohl im internen Format als auch durch normales Lesen der Insatnzen vom Dictionary über PI's zugegriffen werden. Dabei ist der Zugriff über das interne Format schneller, der Zugriff über diPI-Handle jedoch vollständiger, da intern nicht alle Informationen übernommen werden.

Beispiel Um die Structure-Definition im internen Format bereitzustellen, können Dictionary-Funktionen verwendet werden:

```
    StructDef  *str_def;
    Dictionary
dict(„d:\\ode\\sample\\sample.res",PI_Read);
    str_def = dict.ProvideStructDef(„Person");
```

Genauso ist es jedoch auch möglich, die Structure-Definition als Instanz des ODABA2-Objektmodells zu bereitzustellen:

```
    SDB_Structure    *str_def;
    Dictionary        dict(„d:\\ode\\sample\\sample.res",
                           PI_Read);
    PI(SDB_Structure)                          str_pi(dict,"
SDB_Structure",PI_Read);

    str_def = str_pi->Get(„Person");
```

Diese beiden Varianten unterscheiden sich zum einen im Format (einmal wird die Structure-Definition als **StructDef**, im anderen Fall als SDB_Structure-Instanz bereitgestellt), zum anderen aber auch in ihrem Informationsgehalt, da die interne Structure-Definition nur die zum Zugriff erforderlichen Informationen enthält.

Ein geöffnetes Dictionary ist Grundvoraussetzung für das Bearbeiten einer Datenbank. Sind Schema- und Sachebene in einer Datenbank gespeichert, ist es ausreichend, nur das Dictionary zu öffnen, da das Dictionary ja gleichzeitig ein Datenbank-Handle ist und somit den Zugriff auf beliebige Daten der Datenbank ermöglicht.

4.1.2 System Class DBHandle (Datenbank-Handle)

Das Datenbank-Handle vermittelt die Zugriffe auf die in der Datenbank gespeicherten Structure-Instanzen. Die Structure-Definitionen zu den Daten befinden sich in dem Dictionary, das mit dem Datenbank-Handle verbunden ist. Damit ist es prinzipiell möglich, eine Dictionary mit einer beliebigen Datenbank zu verbinden bzw. für eine Vielzahl von Datenbanken einzusetzen.

Beispiel Um ein Datenbank-Handle zu konstruieren, muß vorher ein Dictionary geöffnet worden sein. Das kann in folgender Form geschehen:

```
   Dictionary
dict(„d:\\ode\\sample\\sample.res“,PI_Read);
   DBHandle    db_handle(&dict,
                         „d:\\ode\\sample\\sample.dat“,
                         PI_Write);
```

In diesem Fall fungiert die „SAMPLE.RES“ als Dictionary für die Datenbank „SAMPLE.DAT“.

Auf der Ebene des Datenbank-Handles sind auch die Transaktionen mit ihren Instanzenpuffern angesiedelt. Mit dem Öffnen oder Konstruieren eines Datenbank-Handles wird eine Datenbanktransaktion gestartet, die jedoch im Unterschied zu untergeordneten Transaktionen die geänderten Instanzen sofort in der Datenbank speichert.

Ebenfalls auf der Datenbankebene angesiedelt ist die LOG- und Recovery-Datei, die geschrieben werden kann, um Widerherstellungsprozesse zu aktivieren oder ein Nutzerbezogenes Änderungsprotokoll zu erstellen.

Ein geöffnetes Datenbank-Handle ist Voraussetzung für den Zugriff auf die Real Objects und die Structure-Instanzen in einer Datenbank. Da das Datenbank-Handle gleichzeitig ein Handle für das Root Object ist, das in jeder Datenbank beim Anlegen automatisch erzeugt wird, ist das explizite Öffnen eines Object Handles nur erforderlich, wenn auf untergeordnete Real Objects zugegriffen werden soll.

4.1.3 System Class ACObject (Object Handle)

Das Object Handle vermittelt den Zugriff zu den Instanzen eines Real Objects. Da allen Instanzen ein beobachtendes Objekt zugrundeliegt, sind die Instanzen über Extents einem Real Object, mindestens dem Root Object, zugeordnet. Dadurch ist es möglich, in einer Datenbank Sichten mehrere Subjekte zu speichern.

Beispiel Faßt man die Projekte, die in einer Firma bearbeitet werden, als Real Objects auf, können die Projektressourcen in einer Datenbank gespeichert werden, indem sie verschiedenen Real Objects zugeordnet werden. Um ein Project einzustellen, muß dann nur ein Object Handle für das entsprechende Projekt geöffnet werden.

```
   Dictionary
dict(„d:\\ode\\sample\\sample.res“,PI_Read);
   DBHandle    db_handle(&dict,
                         „d:\\ode\\sample\\sample.dat“,
                         PI_Write);
   ACObject
object_handle(&db_handle,„Sample1“,PI_Write);
```

Da untergeordnete Real Objects wieder untergeordnete Real Objects besitzen können, können mit dem Object Handle nun weitere unter-untergeordnete Object Handle erzeugt werden.

Real Objects können beliebig tiefe Hierarchien bilden, so daß weitere Object Handle geöffnet werden können, sollte dies erforderlich sein. Real Objects werden über Konstruktoren erzeugt und persistent gespeichert. Die Zuordnung von Extents zu einem Real Object erfolgt dynamisch in dem Moment, in dem Daten zu einem Extent in dem Object gespeichert werden sollen. Die meisten Beziehungen (z.B. alle Extent-basierten Relationships) werden im Kontext eines Real Objects gepflegt. Es können jedoch auch Beziehungen üner Object-Grenzen hinaus hergestellt und gespeichert werden.

4.1.4 System Class PI (Persistent Instance Handle)

PI-Handle sind generische Klassen, die entsprechend dem Type der zu bearbeitenden Instanzen erzeugt werden. Über PI-Handle wird der Bezug auf die

Instanzen eines Extents oder einer freien Collection in einem Real Object hergestellt.

Beispiel Um auf die PERSONEN zuzugreifen, die im Root Object der SAMPLE.DAT-Datenbank gespeichert sind, kann eine PERSONEN-PI erzeugt werden:

```
   Dictionary
dict(„d:\\ode\\sample\\sample.res“,PI_Read);
   DBHandle    db_handle(&dict,
                         „d:\\ode\\sample\\sample.dat“,
                         PI_Write);
   PI(Person)  pers_pi(&db_handle,„Person“,PI_Write);
```

Da das Datenbank-Handle gleichzeitig ein Object Handle für das Root Object ist, ist das explizite Öffnen eines Object Handles nicht erforderlich.

PI-Handle sind mit einer umfassenden Funktionalität ausgestattet, die vielfältige Operationen auf die Instanzen ermöglichen. Für Instanzen, die über PI-Handle bereitgestellt werden und References auf andere Structure-Instanzen enthalten, werden beim bereitstellen die Datenbankreferenzen durch PI-Handle ersetzt. Die folgende Tabelle zeigt die Darstellung der einer PERSONEN-Instanz und die entsprechende Definition auf der Schema und Modellebene:

Attributes			**Relationships**		
Name	***Geburtsdatu***	....	**Mitarbeiter**	**Kinder**	**Eltern**
Paul Müller	*1.6.1953*	....	PI(Mitarbeiter)	PI(Kinder)	PI(Eltern)

In der Darstellung internen Instanz werden Datenbankreferenzen für Mitarbeiter, Kinder und Eltern durch die entsprechenden PI-Handle ersetzt. Mit diesen PI-Handles kann wie über Pointer direkt auf nachgeordnete Instanzen zugegriffen werden.

Beispiel Ein MITARBEITER ist einer Abteilung zugeordnet, es gibt also eine Relationship zur ABTEILUNG, die in der MITARBEITER-Instanz als Reference auf eine ABTEILUNGS-Instanz dargestellt wird. Nach dem Einstellen auf eine MITARBEITER-Instanz wird die ABTEILUNGS-Reference durch ein PI-Handle ersetzt, das den unmittelbaren Zugriff auf die ABTEILUNGS-Instanz erlaubt.

```
   char        *abt_name;
   .....
   PI(Mitarbeiter)
ma_pi(&db_handle,„Mitarbeiter“,PI_Write);
```

```
   abt_name      =      ma_pi(„Müller“)->get_Abteilung()-
>get_Name();
```

Die erste **get**-Funktion liefert dabei das PI-Handle für Abteilung in der eingestellten MITARBEITER-Instanz. PI-Handle können wie Zeiger auf die referenzierte Instanz verwendet werden, so daß die zweite **get**-Funktion den Namen aus der referenzierten ABTEILUNGS-Instanz bereitstellt.

Persistente Instanzen müssen nicht unbedingt in der ODABA2-Datenbank gespeichert sein. ODABA2 kann über PI-Handle auch auf Instanzen zugreifen, die in anderen Datenbanken gespeichert sind, z.B. in Relationen, auf die über ODBC zugegriffen werden kann oder in einfachen Dateien (Binär-Dateien, Textdateien im SDF- oder OEL-Austauschformat).

PI-Handle sind Iteratoren, d.h. sie können nicht nur zum Zugriff auf einzelne Instanzen verwendet, sondern auch zur iterativen Verarbeitung von Collections benutzt werden. Dabei können Kopien zu einem PI-Handle erzeugt werden, so daß eine Collection gleichzeitig über mehrere Iteratoren verarbeitet werden kann.

4.1.5 System Class DBField (Property)

DBFields ermöglichen den Property-bezogenen Zugriff auf Felder einer Datenbank. Dadurch wird zum einen der feldbezogene Zugriff vereinfacht, da Datenkonvertierungen implizit vorgenommen werden und verschiedene Operationen zwischen DBFileds unterstützt werden.

Beispiel Um einem MITARBEITER ein neues Gehalt zuzuweisen, sind normalerweise Kenntnisse über die Speicherform (**int** oder **float**) des Feldes erforderlich. Bei der Verwendung von DBFields findet in jedem Fall die Zuweisung im richtigen Format statt.

```
   PI(Mitarbeiter)
ma_pi(&db_handle,„Mitarbeiter“,PI_Write);
   DBField         gehalt(ma_pi,“Gehalt“);

   gehalt = 4510.50;
```

Dabei wird nicht nur die korrekte Konvertierung gewählt, sondern auch eine Modifikationsanzeige für die Instanzenänderung gesetzt, so daß später ein Sichern der Instanz erfolgen kann.

Für die meisten Datentypen werden für DBFields Zuweisungen, Vergleichsoperatoren und arithmetische Operationen unterstützt. Darüberhinaus können aber Property-Descriptoren und andere Schemainformationen über das DBField bereitgestellt werden. Für Reference Properties besitztzen DBFields außerdem alle Eigenschaften eines PI-Handles. In diesem Sinne können auch generische DBFields erzeugt und verwendet werden.

Beispiel Um das Durchschnittsalter der **Kinder** einer PERSON zu ermitteln, kann ein generisches DBField erzeugt und als Iterator verwendet werden:

```
   PI(Person)
pers_pi(&db_handle,„Mitarbeiter",PI_Write);
   DBField       kinder(pers_pi,"Kinder");
   DBField       alter(kinder,"Alter");
   DBField       summe("INT",4);
   short         anzahl = 0;

   kinder.Cancel();
   while ( kinder++ )
   {
      summe += alter;
      anzahl++;
   }
   summe = summe/anzahl;
```

Durch den PI-Operator ++ für das DBField **kinder** wird die jeweils nächste Instanz der Kinder-Collection eingestellt. Dann wird das DBField **alter** aus der Kinder-Instanz auf das interne DBField **summe** addiert. Dabei erforderliche Konvertierungen werden vom DBField erkannt und automatisch durchgeführt.

Die interne Darstellung der Property-Deskriptoren erfolgt in einem **fmcb**, der die wesentlichen Attribute einer Property enthält. Außerdem sind DBFields mit einem Kontext verbunden, so daß bei Modifikationen eines DBFields entsprechende Events erzeugt und Gültigkeitsprüfungen ausgeführt werden.

Applikationen, die vollständig auf der Basis von DBFields entwickelt werden, haben den Vorteil, daß sie Schemaänderungen gegenüber unempfindlich sind. Bei Änderungen der Structure-Definitionen sind also keine Übersetzungen erforderlich, solange keine neuen Leistungen implementiert werden sollen.

4.2 Implementierungsklassen für Problem Structures

Problem Classes sind alle die Klassen, die das Verhalten problemrelevanten Structures als Methoden darstellen. Methoden für Problem Structures kennen nur die Structure-Instanz und die mit ihr über References verbundenen Instanzen. Eine Structure-Instanz hat in diesem Sinne kein Wissen darüber, in welchem Zusammenhang sie gerade betrachtet wird. Das hat den Vorteil, daß Methoden von Problem Classes beliebigen Zusammenhängen eingesetzt werden können.

ODABA2 unterstützt in seinem erweiterten Entwicklungsmodell Modell die Implementierung verschiedener Methodenkategorien.

- **Programm-Funktionen**

  Dies ist die verbreitetste Form der Implementierung von Methoden. Das ODABA2-Entwicklungsmodell unterstützt derzeit die Implementierung von Methoden in C++-Klassen, ist aber für weitere Programmiersprachen offen.

- **Expressions**

  Eine einfache, aber etwas eingeschränkte Form der Implementierung stellen OQL-Expressions dar. Über diese ist sowohl der Zugriff als auch die Manipulation von Structure- und Property-Instanzen möglich.

- **Templates**

  Eine weitere Möglichkeit besteht in der Implementierung von Methoden als Templates. Dies sind Muster (z.B. für eine Maske oder ein Dokument), die für bestimmte Methodenkategorien bereitgestellt werden können.

4.2.1 Program Function

Program Functions sind Funktionen, die in ODABA2 als C++-Funktionen implementiert werden. Die Implementierung von Methoden als Funktionen in einer Implementierungsklasse ist zwar der aufwendigste, aber auch der effizienteste Weg.

Sie erfolgt, indem Structure-Definitionen (→ **SDB_Structure**) auf der Schemaebene zu Implementierungsklassen (→ **ODC_ImpClass**) spezialisiert werden. Dabei werden die Properties der Structure einschließlich der Ableitungshierarchie an die Implementierungsklasse vererbt, d.h. die Klassenhierarchie entspricht vollständig der Structure-Hierarchie.

Wie die Eigenschaften der Structure werden auch die Beschreibungsobjekte vererbt, so daß mit dem Erzeugen einer Klasse bestehende Beschreibungsobjekte erhalten bleiben, jedoch durch zusätzlich Beschreibungsobjekte für die Methoden ergänzt werden. Die vielfältige Darstellung und Unterstützung von Zusammenhängen zwischen Implementierungsklassen und anderen Entwicklungsobjekten ist im Abschnitt „ODE - Die ODABA2 Entwicklungsumgebung“ detailliert beschrieben.

Implementierungsklassen können nicht nur für Datenbankstrukturen, sondern auch als interne (transiente) Klassen einer Anwendung definiert werden. Auf diese Weise kann das Objektmodell, daß i.allg. vorerst die persistenten Structures enthalten wird, zunehmend um transiente Structures erweitert werden.

4.2.2 Expressions

Expressions sind Funktionen einer Objektklasse, mit deren Hilfe abgeleitete Werte gebildet werden können. Diese abgeleiteten Werte werden als Property-Instanzen bereitgestellt. In diesem Sinne werden Expressions vor allem für die Bereitstellung von Werten in transienten Feldern oder zur Spezifikation von Gültigkeitsbedingungen verwendet.

Expressions werden in OQL-Classes (→ **OQL_Class**) abgelegt, die eine Spezialisierung von Structure-Definitionen darstellen. Damit erben auch OQL-Classes die Properties und Ableitungshierarchie ihrer Structures. An ODABA2-Expressions können zum einen Parameter übergeben, zum anderen können aber auch lokale Variablen berechnet werden. Über eine standardisierte Fehlerbehandlung können sowohl Zugriffsfehler als auch inhaltliche Fehler abgefangen werden.

Beispiel Um das Durchschnittsalter der **Kinder** einer PERSON zu ermitteln, kann ein OQL-Expression definiert werden:

```
REAL Person::KD_Alter()
{
   k_anzahl = Kinder.GetCount();
   summe    = Kinder.Sum(Year(Date()-Geburtsdatum));

 BEGINSEQ

   if ( k_anzahl == 0 )
     ERROR(111)
    else
     summe/k_anzahl;

 RECOVER (YES)

   -1;
}
```

Der eigentliche Expression ist zwischen der BEGINSEQ und der RECOVER-Anweisung definiert. Im Fehlerfall (Zugriffsfehler oder Anzahl der Kinder ist 0) wird der Expression im RECOVER-Block ausgeführt, wobei eine Nachricht über den aufgetretenen Fehler erzeugt wird. Der RECOVER-Block wird auch aktiviert, wenn bei der Berechnung der lokalen Variablen ein Fehler auftritt (z.B. wenn der Expression ohne Instanz aufgerufen wird).

In Expressions können andere Expressions oder definierte Aktions aufgerufen werden. Damit können nicht nur Kontextfunktionen ausgeführt, sondern auch Dialoge aufgerufen oder Dokumente erzeugt werden. Obwohl Experessions normalerweise keine Änderungen an Instanzen ausführen können, ist es möglich, daß durch aufgerufene Kontextfunktionen Änderungen veranlaßt werden. In diesem Sinne können also auch Expressions Nebeneffekte auslösen.

Expressions werden in ODABA2 interpretierend verarbeitet. Daduch sind sie sehr flexible im Einsatz, da Änderungen an Expressions sofort wirksam werden. Sie sind jedoch auch am langsamsten in der Auswertung. Expressions werden direkt als Methoden in der Ressourcendatenbank gespeichert. Damit können sie auch zum Ausführungszeitpunkt noch geändert werden.

4.2.3 Template Classes

Die Technik der Template Methoden basiert auf der Annahme, daß es Methoden gibt, die typischen Mustern folgen. Wie die Methoden, die in einer Programmiersprache implementiert werden, werden Template Methoden in entsprechenden Template Classes implementiert. Allerdings werden Templates Methoden nicht in Form einer Sprache mit definierter Syntax spezifiziert, sondern als Muster (Window, Dokument) gestaltet. ODABA2 unterstützt drei Arten von Template-Methoden.

- **View**

  View Templates oder Views sind vordefinierte Sichten zur Darstellung komplexer funktionaler Zusammenhänge. Sie gehen über die Definition von OQL-Views hinaus, da sie als persistente Views implementiert und auf Events reagieren können.

- **Window**

  Window Templates oder Windows stellen Daten zur Bearbeitung oder Ansicht im Rahmen einer graphischen Benutzeroberfläche bereit. Im Gegensatz zu den meisten Masken-Editoren der Programmierumgebungen sind Window Templates mit dem Datenhintergrund verbunden.

- **Document Templates**

  Document Templates beschreiben die Darstellung von Daten in Dokumenten. Dokumente können Tabellen, Kapitelstrukturen, aber auch Programme u.a.m. sein.

4.2.3.1 Views Templates

Eine View definiert eine bestimmte Sicht auf einen Zusammenhang. Diese Sicht bildet einen speziellen funktionalen Zusammenhang ab und stellt ihn als Menge von View Instanzen dar. Views können eine relationale Sicht auf objektorientierte Darstellungen erzeugen und sind ein geeignetes Mittel zur Kommunikation über allgemeine Schnittstellen wie z.B. ODBC (Open Database Connectivity) für Microsoft Windows.

Views Templates, hier kurz Views, sind Methode Templates zur Darstellung von Views. Als solche stellen sie sowohl einen strukturellen als auch einen methodischen Sachverhalt dar. Hinzu kommt, daß einige Aspekte einer View auch im Zustandsmodell dargestellt werden können, was zu weiteren Möglichkeiten bei der Definition von Views führt. Die Definition einer View erfolgt im Objektmodell. Da jedoch die Bildung der Property-Instanzen der View im wesentlichen auf Expressions beruht, ist die View sehr eng mit den entsprechenden Methoden verbunden, d.h. eine View bildet durchaus auch methodische Aspekte ab. In diesem Sinne wird die View in ODABA2 in gleicher Weise als Methode als auch als struktureller Zusammenhang aufgefaßt und dargestellt. Eine genauere Beschreibung der View erfolgte bereits bei der Darstellung des Objektmodells. Aus diesem Grunde soll hier auf weitere Details versichtet werden.

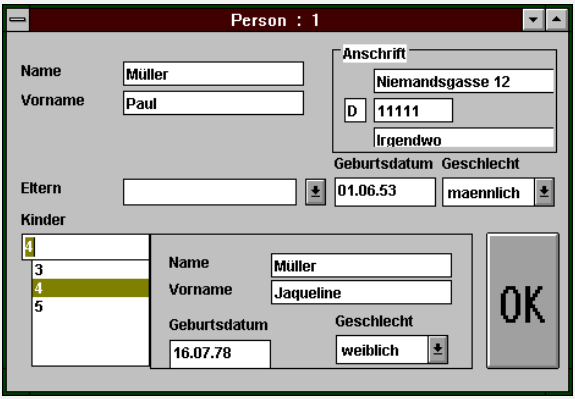
4.2.3.2 Window Templates

Window Templates sind Methoden zur Darstellung von Structure-Instanzen im Rahmen einer graphischen Oberfläche. Dabei können in gleicher Weise Structure-Instanzen und View-Instanzen dargestellt werden. Ebenso ist es möglich, freie Collections in unterschiedlicher Form (Schlüsselliste, Tabelle) anzuzeigen.

Windows werden als Methoden in Window-Klassen (→ **Win_Class**) implementiert, die Spezialisierungen von Structures sind. In diesem Sinne erben auch die Window-Klassen die Eigenschaften der Structure und ihre Beschreibungen. Windows können wie andere Methoden in Expressions oder an anderen Stellen aufgerufen werden, vorausgesetzt, diese werden in einer Umgebung ausgeführt, die das Anzeigen von Windows erlaubt.

In einem Window können die Property-Instanzen einer Structure in entsprechenden Controls angezeigt werden. Die Vielfältigkeit der Property-Arten (References, Base Structures, Structured Attributes, Array Attributes usw.) unterstützen die Window Templates durch die Definition von Controls mit entsprechenden Leistungsmerkmalen. Die Definition von Window Templates basiert auf der Structure-Definition der Window-Klasse. Die Controls werden mit den entsprechenden Property-Instanzen verknüpft, so daß sie beim Anzeigen des Windows automatisch mit den Daten der Instanz gefüllt werden können.

Beispiel Die Darstellung einer PERSONEN-Instanz in einem Window veranschaulicht einige der Möglichkeiten der Darstellung verschiedener Property-Arten:



Um die Daten der referenzierten Structure-Instanzen anzuzeigen, gibt es Möglichkeiten, nachgeordnete Window Templates zu definieren, die bei Bedarf geöffnet werden können. Diese Leistungen werden durch die Standardfunktionalität der Window-Klassen möglich und bedürfen keiner weiteren Programmierung. Property-Instanzen können über ein Control geändert, Structure-Instanzen zu einer Reference oder freien Collection hinzugefügt oder gelöscht werden und vieles andere mehr.

Die Verbindung eines Windows mit einer Structure-Instanz stellt einen neuen Zusammenhang dar, in dem verschiedene Ereignisse eintreten können, die auf der Datenbankebene unbekannt sind. So kann das Aktivieren eines Windows oder eines Controls spezielle Reaktionen erforderlich machen, die auf der Datenbankebene nicht darstellbar sind. Diese speziellen Zusammenhänge können wieder über entsprechende Context Classes dargestellt werden (mehr dazu in der Dokumentation „ODE - Die ODABA2 Entwicklungsumgebung“).

4.2.3.3 Document Templates

Document Templates sind Methoden zur Darstellung von komplexen Zusammenhängen in einem Dokument. In einem Document Template können weitere Templates oder Methoden zur Bereitstellung abgeleiteter Daten oder zur Darstellung untergeordneter Instanzen aufgerufen werden. In diesem Sinne verhalten sich Document Templates ähnlich wie Funktionen einer Programmiersprache oder Expressions. Die Definition von Document Templates basiert auf einer View- oder Structure-Definition der Schemaebene, d.h. ein Document Template ist eine Methode der von der entsprechenden Structure abgeleiteten Template Class.

In diesem Sinne werden Documents als Methoden in Document-Klassen (→ **Doc_Class**) implementiert, die Spezialisierungen von Structures sind. Somit erben auch die Document-Klassen die Eigenschaften der Structure und ihre Beschreibungen. Documents können wie andere Methoden in Expressions oder anderen Methoden aufgerufen oder als Actions definiert werden.

Das Prinzip der Document Templates besteht darin, daß das Dokument vom Anwender in der gewünschten Form als Muster des entsprechenden Textverarbeitungssystems vorbereitet wird, wobei an Stellen, an denen Variablen eingefügt werden sollen, entsprechende OQL-Expressions eingesetzt werden. Diese werden bei der Erstellung des Dokumentes berechnet und in das Dokument eingefügt. Durch Schachtelung der Document Templates können beliebig komplexe Dokumente erstellt werden, Tabellen und andere Auswertungen eingeschlossen. Der Anwendungsbereich für Document Templates reicht von der Source Code-Generierung über die Erstellung von Tabellen und Dokumentationen bis zur Generierung von Online-Hilfen. Durch die Verwendung von Document Templates ist die Gestaltung des Dokuments im Rahmen des Textverarbeitungssystems völlig frei.

Beispiel Das folgende Beispiel zeigt, wie die Kinder von PERSONEN in einer Tabelle angezeigt werden können. Konstante Texte werden wie normaler Text angegeben. Expressions und Property-Bezüge stehen in Klammern **«»**. Relevante Teile der Template-Definition sind durch .DEFINE und .END eingeschlossen. Vor der .DEFINE- und nach der .END-Anweisung können Kommentare untergebracht werden.

.DEFINE DOCUMENT Person.KinderListe STATIC BEGIN

Diese Liste enthält alle Eltern und deren Kinder. Dabei werden die Kinder jeweils beim Vater und bei der Mutter aufgelistet. Personen ohne Kinder werden nicht angezeigt.

«Person().ShowPerson()»

.END

Nach dem einleitenden Text wird das PERSON-Template **ShowPerson**() für jede PERSON des Extents **Person** aufgerufen.

.DEFINE UNIT Person.ShowPerson CONDITION(Kinder.cardinality() > 0) BEGIN

Name : *«Name»* **Vorname :** *«Vorname»*
geb.: «Geburtsdatum»
Anschrift : «Adresse.Strasse», «Adresse.PLZ» «Adresse.Ort»

Kinder:

Name, Vorname	**Geburtsdatum**	**zweites Elternteil**
«Kinder().TabZeile(Ehegatte)»		
Durchschnittsalter :	«D_Alter()»	

.END

Durch die CONDITION in der .DEFINE-Anweisung wird verhindert, daß Personen ohne Kinder in dem Dokument angezeigt werden. Nach den Angaben zur Person wird eine Tabelle mit je einer Zeile pro Kind erzeugt. Dazu wird für alle Kinder das Zeilen-Template **TabZeile**() der Structure PERSON aufgerufen, da die **Kinder**-Instanzen wieder PERSONEN-Instanzen sind. Diesem Template wird die PERSONEN-Instanz des Ehegatten als Parameter übergeben. Abschließend wird durch die Methode **D_Alter**() das Durchschnittsalter der Kinder berechnet und in der letzten Zeile eingetragen.

.DEFINE UNIT Person.TabZeile(Person partner) BEGIN

«Name», «Vorname»	«Geburtsdatum»	«partner.Name», «partner.Vorname»

.END

Durch dieses Template werden in der Zeile Name, Vorname, Geburtsdatum und Angaben zum zweiten Elternteil eingetragen. Die Ausführung des Document Templates liefert dann folgendes Dokument:

Diese Liste enthält alle Eltern und deren Kinder. Dabei werden die Kinder jeweils beim Vater und bei der Mutter aufgelistet. Personen ohne Kinder werden nicht angezeigt.

Name : *Müller* **Vorname :** *Martha*
geb.: 2.5.1952
Anschrift : Niemandsgasse 12, 11111 Irgendwo

Kinder:

Name, Vorname	Geburtsdatum	zweites Elternteil
Müller, Anton	5.3.1976	Müller, Paul
Müller, Ernesto	11.11.1980	Müller, Paul
Müller, Jacqueline	16.7.1976	Müller, Paul
Durchschnittsalter :	17	

Name : *Müller* **Vorname :** *Paul*
geb.: 1.6.1953
Anschrift : Niemandsgasse 12, 11111 Irgendwo

Kinder:

Name, Vorname	Geburtsdatum	zweites Elternteil
Müller, Anton	5.3.1976	Müller, Martha
Müller, Ernesto	11.11.1980	Müller, Martha
Müller, Jacqueline	16.7.1976	Müller, Martha
Durchschnittsalter :	17	

Die Definition der Document Templates erfolgt zur Zeit im RTF-Format (Rich Text Format). Dadurch können Document Templates in verschiedenen Textverarbeitungssystemen erzeugt werden. Die später generierten Dokumente werden dann ebenfalls im RTF-Format bereitgestellt, so daß auch sie durch verschiedene Textverarbeitungssysteme weiterverarbeitet werden können. Auf diese Weise können die Templates auf viele Möglichkeiten des Textverarbeitungssystems wie Schriftarten, Steuerung für Seitenwechsel, Kopf- und Fußzeilen u.a.m. zurückgreifen.

4.3 Context Classes

In vielen Fällen reichen die durch die Problem und System Classes bereitgestellten Methoden nicht aus, da sie den Zusammenhang nicht kennen, in dem eine Instanz dargestellt wird. Dieser Zusammenhang kann darin bestehen, daß eine Structure-Instanz im Zusammenhang einer Collection-Instanz gespeichert wird, deren Element sie ist, oder in einem Window auf dem Bildschirm angezeigt wird. Nicht selten findet aber gerade in dem konkreten Umfeld eine Spezialisierung des Verhaltens der Instanz, eine Rollenspezialisierung, statt. Damit wird dieses Umfeld, der Context, zu einem entscheidenden Faktor für die korrekte Arbeitsweise einer Methode.

> **Beispiel** Wenn bezüglich der Menge der PERSONEN Teilmengen für **Männer** und **Frauen** gebildet werden, kann beim Erzeugen von PERSONEN-Instanzen das **Geschlecht** korrekt initialisiert werden. Wird eine Instanz über die Collection **Männer** erzeugt, soll das **Geschlecht** auf *männlich*, im Zusammenhang mit der Collection **Frauen** auf *weiblich* initialisiert werden. Da eine Methode **initialize()** von PERSON nichts über diesen Zusammenhang weiß, wird sie diese Anforderung nicht umsetzen können. Erst durch eine Context Class, die den konkreten Zusammenhang kennt, können spezialisierte Initialisierungsfunktionen bereitgestellt werden, so daß die korrekte Initialisierung für **Männer** und **Frauen** ohne zusätzliche Abfrage möglich ist.

Context Classes stellen die Verbindung zwischen Instanzen und einem bestimmten Umfeld, ihrem Context her. Dabei hat jeder Context seine eigenen zusätzlichen Informationen, die eine Spezialisierung des Verhaltens im Rahmen dieses Zusammenhangs erlauben.

Contexts entstehen auf verschiedenen Ebenen, so daß dementsprechend auch Context Classes auf verschiedenen Ebenen gebildet werden:

- **Datenbank**

  Contexts auf der Ebene der Datenbank beschreiben Zusammenhänge, die nur durch die Darstellung in der Datenbank festgelegt sind. Context Classes können hier für alle Ebenen (Datenbank, Real Object, Structure Instanz, Property) gebildet werden.

- **GUI (Gaphische Benutzeroberflächen)**

  Im Zusammenhang mit graphischen Benutzeroberflächen entstehen weitere Context Classes, die sich aus dem Zusammenhang von Datenbankinstanzen und Öberflächenelementen (Project, Application, Window, Control) ableiten.

- **Dokument**

  Document Contexte beschreiben das besondere Verhalten einer Structure Instanz in einem Dokument. Hier bezieht sich das kontextspezifische Verhalten vor allem auf die Auswahl von Instanzen und Spezifikation der Ausgabedateien.

Context Classes beschreiben also nicht nur strukturelle Zusammenhänge in der Datenbank, sondern auch Zusammenhänge, die mit der Darstellung von Instanzen verbunden sind. Da die Komplexität der dargestellten Zusammenhänge einer ständigen Entwicklung unterliegt, sind in der Zukunft durchaus weitere Zusammenhänge denkbar, die in die Darstellung über Context Classes einbezogen werden.

4.3.1 Allgemeine Eigenschaften von Context Classes

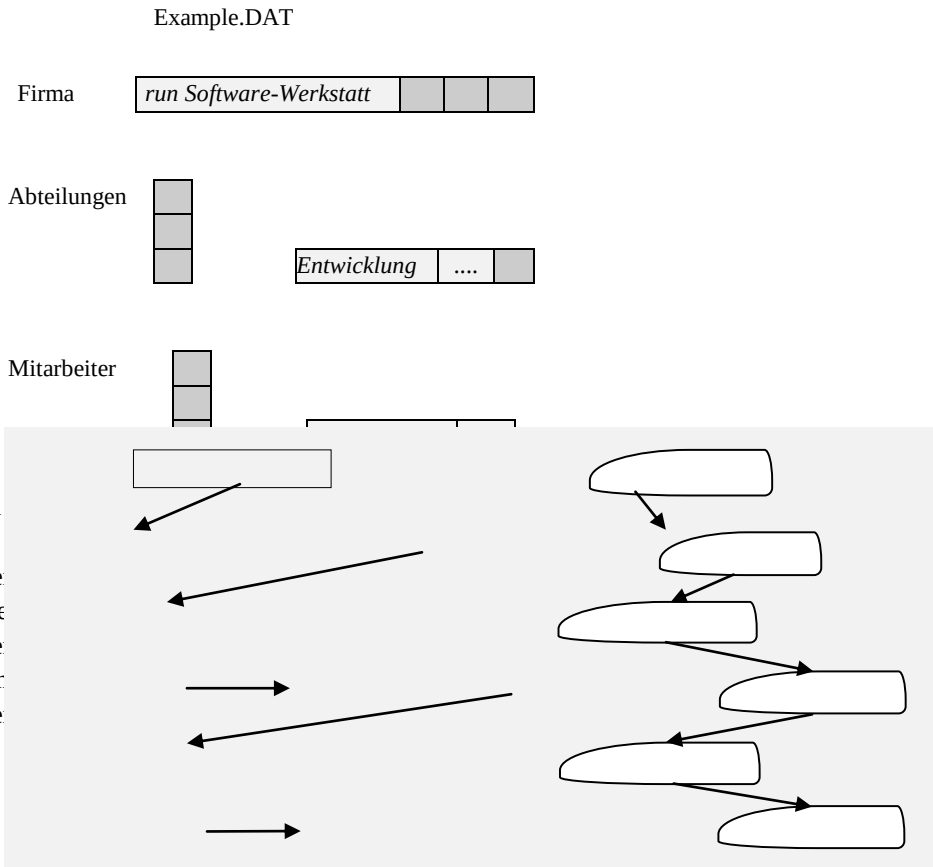
Im folgenden sollen einige allgemeine Eigenschaften von Context Classes dargestellt werden, die in gleicher Weise auf die verschiedenen Context Classes zutreffen.

Context-Hierarchien

Contexte bilden Hierarchien, wie am Beispiel der Datenbank-Context Classes leicht zu sehen ist. So wird z.B. der Context einer Structure oder Property-Instanz durch eine Instanz gebildet, die wieder einen übergeordneten Context besitzt, d.h. ein Property Context wird in einem Structure Context dargestellt. Die Structure-Instanz ist wiederum vielleicht Element einer Reference und wird somit im Collection Context dargestellt. Auf diese Weise entsteht eine Context-Hierarchie, die durch all die Collections und Instanzen gebildet wird, die den Weg zu einer Property oder Structure-Instanz in einem konkreten Umfeld beschreiben. Ausgangspunkt dabei ist

meistens ein Real Object Context, der seinerseits einen weiteren übergeordneten Real Object oder einen übergeordneten Data Base Context besitzen kann. In diesem Sinne werden Context-Hierarchien durch Instanzenpfade beschrieben, die den Weg aufzeigen, der in einem bestimmten Zusammenhang zu einer Instanz führt. Da der Weg zu einer Instanz nicht immer eindeutig ist, sind auch die Context-Hierarchien nicht immer eindeutig bestimmt, d.h., daß ein und dieselbe Instanz in verschiedenen Contexts erscheinen kann.

Beispiel Wird eine MITARBEITER-Instanz bereitgestellt, ist immer ihr Structure Context aktiv. Wurde sie als Instanz der **Mitarbeiter**-Collection der ABTEILUNG bereitgestellt, ist der übergeordnete Context ein Collection Context für **Mitarbeiter** der ABTEILUNGEN. Dieser wiederum ist eine Reference in der Structure ABTEILUNG, so daß der übergeordnete Context der Collection wieder ein Structure Context für die ABTEILUNGS-Instanz ist. Die ABTEILUNGS-Instanz wurde vielleicht im Rahmen eines Extents, also im Collection Context, bereitgestellt, dem der Object Context der FIRMA übergeordnet ist.



Ein
der
übe
eine
übe
Cor
übe

Applikation untergeordnet ist. Da die Application wiederum dem Project untergeordnet ist, ist dem Application Context der entsprechende Project Context übergeordnet.

Die meisten Contexts besitzen wieder übergeordnete Contexts, so daß der Context nicht nur Informationen über das Umfeld, sondern auch über das Umfeld des Umfeldes usw. liefert. Über den Context ist also der gesamte Zusammenhang sichtbar, in dem eine Structure oder Property-Instanz dargestellt wird. Da die übergeordneten Contexts jederzeit erreichbar sind, braucht dieser Zusammenhang nicht vom Anwendungsentwickler dargestellt zu werden.

Neben der Hierachie zwischen den Kontexten einer Kategorie (Datenbank-Contexts, Oberflächen-Contexts) gibt es Verbindungen zwischen den Contexts verschiedener Kategorien, die jedoch nicht als Hierarchie dargestellt werden. So sind die meisten Control Contexts mit dem Property Context der Property verbunden, die sie darstellen, ebenso, wie die Windows i.allg. mit dem Structure Context der dargestellten Structure-Instanz verbunden sind.

Aktiver Context

Contexts, also die Zusammenhänge an sich, existieren zu jedem Zeitpunkt. Instanzen für Context Classes werden jedoch nur gebildet, wenn ein Context im Rahmen einer Anwendung aktiv wird. Das geschieht i.allg., wenn die entsprechende Instanz im Rahmen der Anwendung gelesen oder anderweitig erzeugt wird bzw. wenn eine Window oder ein Control im Rahmen einer Anwendung angezeigt wird. Contexts, die auf diese Weise im Rahmen einer Anwendung aktiviert wurden, werden aktive Contexts genannt.

Dabei besitzt jeder aktive Context im Rahmen seiner Kategorie genau einen übergeordneten aktiven Context, den **High Context**. Der High Context eines aktiven Contexts ist der aktive Context, in dessen Rahmen der betrachtete Context aktiviert wurde. Dieses System der Hierarchie aktiver Contexts setzt voraus, daß ein Context immer durch genau einen übergeordneten Context aktiviert wird. Neben den aktiven Contexts, die in einem konkreten Anwendungsfall entstehen, gibt es eine Vielzahl von inaktiven Contexts, die durch die strukturellen

Zusammenhänge in der Datenbank gebildet werden. Bei bestimmten Aktionen kann es auch erforderlich sein, inaktive Contexts zu aktivieren.

Beispiel Wenn *Anton Müller* als **Mitarbeiter** der **Abteilung** *Entwicklung* zugeordnet wird, indem eine Reference auf die ABTEILUNG in *Antons* MITARBEITER-Instanz eingetragen wird, ist auf jeden Fall der Structure Context der MITARBEITER-Instanz und der Collection Context für **Abteilung** in MITARBEITER aktiv. Da *Anton* nun aber auch in der ABTEILUNG *Entwicklung* als **Mitarbeiter** eingetragen werden muß (inverse Reference in ABTEILUNG), wird vom System die ABTEILUNGS-Instanz der ABTEILUNG *Entwicklung* bereitgestellt und somit ihr Context aktiviert.

Context Classes

Für jede Context-Art wird in ODABA2 eine entsprechende Kategorie von C++-Context Classes bereitgestellt, die die Implementierung von Methoden im Context als C++-Funktionen ermöglicht. Diese C++Context Classes sind mit einer Standardfunktionalität ausgestattet, die verschieden Operationen auf der Context-Ebene ermöglichen.

Darüber hinaus können Context Classes auf der OQL-Ebene gebildet werden. Jede Structure und jede Property läßt die Spezialisierung zu einer OQL-Class zu, in deren Rahmen OQL-Expressions spezifiziert werden können. OQL-Classes können jedoch nur für Structures und Properties, nicht aber für Real Objects oder Datenbanken oder Oberflächen-Context Classes gebildet werden. Es ist jedoch möglich, OQL-Expressions in beliebigen anderen Context Classes zu referenzieren.

Bildung von Context Classes

Die Instanz einer Context Class wird in dem Moment konstruiert, in dem eine Structure oder Property-Instanz im Rahmen einer Anwendung bereitgestellt oder ein Object oder Oberflächenelement zur Bearbeitung eröffnet wird. Im Gegensatz zu Instanzen für Problem Classes, die durch die Anwendung erzeugt werden, werden Instanzen für Context Classes von ODABA2 erzeugt, wenn das entsprechende Objekt und somit ein Context aktiviert wird.

Normalerweise werden Funktionen der Context Classes von ODABA2 aufgerufen, wenn bestimmte Ereignisse - Events - eingetreten sind. Der Sinn der Context Classes besteht also hauptsächlich darin, die Reaktionen auf Events und andere

Zusammenhänge des Zustandsmodells umzusetzen. Context Classes ermöglichen aber auch den direkten Aufruf von Actions, die in verschiedenen Zusammenhängen definiert werden können.

4.3.2 Datenbank-Context Classes

Datenbank-Context Classes beschreiben rollenspezifisches Verhalten, das sich aus den besonderen Zusammenhängen in einer Datenbank ergibt. Die über Datenbankkontexte dargestellten Zusammenhänge sind in dem Sinne allgemeingültig, als daß sie in jeder Anwendung wirken. Prüfungen, Standardinitialisierungen u.a.m werden also selbst beim Aufruf allgemeiner Service-Funktionen wie z.B. beim Kopieren von Instanzen wirksam. Im Falle einer Client-Server-Anwendung werden die Methoden der Datenbankkontexte im Gegensatz zu GUI-Kontexten immer auf der Server-Seite ausgeführt.

In diesem Sinne legen die Methoden der Datenbankkontexte die allgemeinsten Regeln fest, die im Zusammenhang einer Datenbank und damit für jede Anwendung gelten - sie beschreiben das der Datenbank inhärente Verhalten in C++ oder OQL-Context Classes.

Im Rahmen einer Datenbank ist der Context einer Structure Instanz durch die Property festgelegt, für die eine Structure-Instanz gespeichert ist. Aber auch die Structure-Instanz stellt einen Zusammenhang, einen Context für ihre Properties dar, so daß auch für Structures Context Classes gebildet werden können, die jedoch weitgehend mit der Problem Class identisch ist. Instanzen können somit in verschiedenen Arten von Contexts dargestellt werden. Der Context ergibt sich aus der jeweils übergeordneten Instanz, die den Zusammenhang für die untergeordnete Instanz darstellt.

- **Property Context**

  Der Property Context beschreibt den Zusammenhang zwischen einer Property-Instanz und der oder den für sie gespeicherten Structure oder Attribute Instanzen. Die Property kann dabei ebenso ein Extent wie die Property einer Structure sein.

- **Collection Context**

  Der Collection Context ist ein spezieller Property Context, der den Zusammenhang zwischen der Collection-Instanz und den Structure-Instanzen der Collection herstellt.

- **Structure Context**

  Der Structure Context beschreibt den Zusammenhang zwischen einer Structure-Instanz und den Property-Instanzen, die in der Structure-Instanz gespeichert sind.

- **Object Context**

  Der Object Context stellt den Zusammenhang zwischen einer Object-Instanz, den Sichten auf das Real Object und den freien Collections des Real Objects her.

- **Data Base Context**

  Der Context der Datenbank stellt das besondere Verhalten für Instanzen dar, die in der Datenbank gespeichert sind, also den Zusammenhang zwischen Instanzen und Datenbank allgemein. Das schließt z.B. die Vergabe und Kontrolle von Zugriffsrechten ein.

4.3.2.1 Data Base Context

Der Datenbankkontext ermöglicht die Spezifikation speziellen Verhaltens, das durch den Zusammenhang der Datenbank gegeben ist. So kann über den Datenbankkontext eine datenbankspezifische Zugangskontrolle oder Benutzeranmeldung und Registratur realisiert werden. Ebenso ist es möglich, auf dieser Ebene Schreib- und Lesevorgänge auf der Datenbank zu kontrollieren oder zu registrieren, Änderungsprotokolle zu erzeigen u.a.m. Eine detaillierte Übersicht über die Events, die auf dieser Ebene erzeugt und durch Methoden der Data Base Context Class verarbeitet werden können, ist im Referenzhandbuch für Kontextklassen (Online-Hilfe) enthalten.

Data Base Context Classes können in ODABA2 nur als C++-Klassen implementiert werden. Mit der Unterstützung weiterer Programmiersprachen wie JAVA oder

EIFFEL ist jedoch zukünftig auch die Implementierung in anderen Programmiersprachen denkbar.

4.3.2.2Object Context

Object Context Classes können für jedes Real Object gebildet werden. Im Gegensatz zu Context Classes für Structures, die für alle Instanzen eines Types definiert sind, sind Real Object Contexts konkret auf ein Object bezogen. Im Context von Real Objects kann die Bildung und das Löschen von nachgeordneten Real Objects und von Extents des Objects beeinflußt werden. Ebenso ist es möglich, auf der Ebene eines Objects Zugangskontrollen und Berechtigungsprüfungen durchzuführen. Eine detaillierte Übersicht über die Events, die auf dieser Ebene erzeugt und durch Methoden der Object Context Class verarbeitet werden können, ist im Referenzhandbuch für Kontextklassen (Online-Hilfe) enthalten.

Auch Object Context Classes können in ODABA2 derzeit nur als C++-Klassen implementiert werden. Mit der Unterstützung weiterer Programmiersprachen wird jedoch auch hier die Implementierung in anderen Programmiersprachen denkbar.

4.3.2.3Structure Context

Der Structure Context beschreibt den Zusammenhang zwischen einer Structure-Instanz und ihren Properties. Im Gegensatz zum Object Context beschreibt der Structure Context jedoch keinen konkreten, sondern einen abstrakten Zusammenhang, der für alle Structure-Instanzen eines Types gültig ist. Im Structure Context können Beziehungen zwischen den einzelnen Properties hergestellt, Instanzen initialisiert und abgeleitete Properties berechnet werden. Gleichzeitig erlaubt der Structure Context die Kontolle von Zugriffsberechtigungen auf der Nutzerebene. Im Structure Context kann auf nutzerdefinierte und auf System-Events reagiert weden. Im Structure Context ist z.B. die Definition der Zustandsmenge für eine Structure angesiedelt. Diese kann in übergeordneten Contexts (z.B. im Collection Context) weiter eingeschränkt werden. Genauso

können allgemeine Initialisierungsregeln (Konstruktoren) im Structure Context definiert werden.

Beispiel Der Zusammenhang zwischen dem **Abschluß** und dem **Alter** einer PERSON kann im Structure Context als Gültigkeitsbedingung definiert werden. Damit wird von der Datenbank gesichert, daß keine 13-jährigen Hochschulabsolventen in der Datenbank gespeichert werden können. Diese Prüfung kann bei jeder Änderung einer Structure-Instanz in diesem Context vorgenommen werden.

Eine detaillierte Übersicht über die System-Events, die auf dieser Ebene erzeugt und durch Methoden der Structure Context Class verarbeitet werden können, ist im Referenzhandbuch für Kontextklassen (Online-Hilfe) enthalten.

Der Structure Context ist eine Spezielisierung einer Structure. Er beschreibt eben gerade das Verhalten einer Structure Instanz im Rahmen einer ODABA2-Datenbank. Da dieses Verhalten durch einige zusätzliche Eigenschaften und Verhaltensweisen geprägt ist (z.B. die Kenntnis von Kontext-Hierarchien), wird das Verhalten der Structure oder der Problem Class im Rahmen einer Structure Context Class um diese Eigenschaften und Verhaltensweisen erweitert. Structure Context Classes können in ODABA2 derzeit als C++- oder als OQL-Classes implementiert werden.

Der übergeordnete Context für einen Structure Context ist ein Property Context, der unter bestimmten Bedingungen auch ein Collection Context sein kann. Die Beziehung zum aktiven übergeordneten Property Context wird über den High Context des Structure Contexts hergestellt. Da dieser Zusammenhang wiederum auf der Basis der System Classes hergestellt wird, schließt der Structure Context sowohl die Structure-Instanz als auch ihre Definitionen auf der Schemaebene ein. Somit sind auch zur Laufzeit Informationen über die Properties (Namen, Type und Größe) der Structure verfügbar.

4.3.2.4Property Context

Instanzen einer Structure werden immer im Kontext einer Property gespeichert. Dies ist entweder die Property einer übergeordneten Structure-Instanz oder ein Extent eines Real Objects. Property Contexts sind wie Strucrure Contexts abstrakt, also auf eine bestimmte Menge von Property Instanzen bezogen. Oft gibt es jedoch

in den verschiedenen Zusammenhängen differenziertes Verhalten für gleichartige Instanzen - es findet eine Rollenspezialisierung im Context der Property statt. Diese Rollenspezialisierung kann im Property Context bzw. in der entsprechenden Property Context Class dargestellt werden. Der Property Context erlaubt genauso die Rollenspezialisierung für Properties elementarer Art. Es können somit nicht nur Zustände auf der Structure-Ebene, sondern auch auf der Property-Ebene kontrolliert werden. Dadurch können z.B. Property-spezifische Prüfbedingungen definiert werden.

> **Beispiel** Das **Alter** einer PERSON in Jahren wird i.allg. als ganzzahliger Wert ausgedrückt. Bezüglich des **Alters** einer PERSON können wir davon ausgehen, daß ein Wert größer als 200 falsch ist. Die Zustandsmenge der zulässigen Werte kann also auf 0 bis 200 eingeschränkt werden. Dies gilt jedoch nicht für das **Alter** allgemein, da BÄUME wesentlich älter werden können.

Der Property Context stellt den Zusammenhang zwischen der Property-Instanz und den durch die Property verwalteten Structure- oder Attribute-Instanzen her. Gleichzeitig ist durch den High Context der Bezug zum übergeordneten Structure Context gegeben. Da dieser Zusammenhang auf der Basis der System Classes hergestellt wird, schließt die Property Context Class sowohl die Property-Instanz als auch ihre Definitionen auf der Schemaebene ein. Da Properties immer in einem Zusammenhang, also nie kontextfrei definert sind, besitzt eine Property auch immer genau einen übergeordneten Context. Dabei kann es sich um einen Context ein Structure Context oder um einen Real Object Context handeln, wenn die Property ein Extent ist.

Im Rahmen des Property Contexts können Propertybezogene Prüfungen und Initialisierungen vorgenommen werden. Eine detaillierte Übersicht über die Events, die auf dieser Ebene erzeugt und durch Methoden der Property Context Class verarbeitet werden können, ist im Referenzhandbuch für Kontextklassen (Online-Hilfe) enthalten.

Property Context Classes können in ODABA2 derzeit als C++- oder als OQL-Classes implementiert werden. Mit der Unterstützung weiterer Programmiersprachen wird auch hier die Implementierung in anderen Programmiersprachen möglich.

Collection Context

Der Collection Context ist eine Spezialisierung des Property Context, d.h. jeder Collection Context ist ein Property Context. Die Einordnung in die Context-Hierarchie erfolgt entsprechend den Regeln für Property Contexts. Der Collection Context erlaubt die spezielle Behandlung im Context eines Extents oder einer Reference. So können im Context einer Collection z.B. spezielle Initialisierungen oder Aktionen beim Hinzufügen oder Löschen von Structure-Instanzen ausgeführt werden.

Beispiel So kann im Collection Context des Extents **Männer** dafür gesorgt werden, daß das **Geschlecht** von PERSONEN beim Hinzufügen mit dem korrekten Wert *männlich* initialisiert wird. Andrerseits kann, wenn *Anton* aus der Firma, in der er arbeitet, entlassen und somit aus der Collection der **mitarbeiter** dieser Firma entfernt wird, dafür gesorgt werden, daß sein Lohnbetrag vom **Lohnfond** seiner ehemaligen ABTEILUNG abgezogen wird.

4.3.3 GUI-Context Classes

Durch die Definition des erweiterten ODABA2-Objektmodells entstehen z.B. durch die Definition von Windows in Window-Klassen weitere spezielle Zusammenhänge, die ein spezialisiertes Verhalten erforderlich machen. Für die wesentlichen Entwicklungsobjekte im Zusammenhang mit graphischen Oberflächen werden deshalb spezielle GUI-Context Classes unterstützt, die es ermöglichen, spezielles Verhalten im Zusammenhang mit graphischen Oberflächen zu implementieren:

- **Project Context**

  Der Project Context beschreibt eine Datenbank im Zusammenhang mit einem speziellen Projektes. Damit kann also auch spezielles Verhalten des Projektes dargestellt werden.

- **Application Context**

  Der Application Context beschreibt das spezielle Verhalten eines Real Objects im Rahmen einer Applikation. Der Application Context beschreibt also auch das besondere Verhalten der Applikation.

- **Window Context**

  Window Contexts beschreiben das spezielle Verhalten von Structure-Instanzen in einem Window. Neben Darstellungsfragen können hier z.B. auch Zugriffsrechte geklärt werden.

- **Control Context**

  Control Contexte beschreiben das Verhalten einer Property in einem bestimmten Control. Dieser Context bezieht sich auf spezielle Darstellungsfragen der Property im Control, sowie auf Sicherung der Zugriffsrechte uam.

Im Gegensatz zu Datenbank-Contextes ist das oberflächenspezifische Verhalten in Client-Server-Anwendungen auf der Client-Seite angesiedelt.

4.3.3.1 Project Context

Der Project Context ermöglicht die Spezifikation speziellen Verhaltens, das sich aus dem Zusammenhang eines Projektes ergibt. So kann über den Project Context eine projektspezifische Zugangskontrolle oder Benutzeranmeldung und Registratur realisiert werden. Ebenso ist es möglich, auf dieser Ebene benutzerbezogene Informationen zur Darstellung (z.B. Größe und Position der Arbeitsfläche des Projektes) auszuwerten. Außerdem kann auf Zeitereignisse reagiert werden, so daß im Rahmen eines Projektes bestimmte Aktionen zu definierten Zeitpunkten ausgeführt werden können. Eine detaillierte Übersicht über die Events, die auf dieser Ebene erzeugt und durch Methoden der Project Context Class verarbeitet werden können, ist im Referenzhandbuch für Kontextklassen (Online-Hilfe) enthalten.

In einer ODABA2-Anwendung ist ein Projekt immer mit einer Datenbank verbunden, die die Projektdaten enthält. Dem Project Context ist der entsprechende Datenbank Context bekannt, so daß sowohl auf Datenbank als auch auf Data Base Context-Funktionen zugeriffen werden kann. Project Context Classes können in ODABA2 als C++-Klassen implementiert werden.

4.3.3.2 Application Context

Ein Projekt kann aus einer oder mehrere Applikationen bestehen. Dabei ist eine Applikation i.allg. mit einem Real Object verbunden. Der Application Context ermöglicht die Spezifikation speziellen Verhaltens, das sich im Zusammenhang einer Applikation in einem Projekt ergibt. Wesentlicher Bestandteil von Applikationen sind Menüs und Acceleratoren, über die applikationsspezifische Aktionen angestoßen werden können. Derartige Menü- und Accelerator-Aktionen können neben Window- und Document-Aktionen im Application Context definiert werden. Damit ist der Application Context Ausgangspunkt der gesamten Funktionalität einer Applikation. Darüber hinaus können benutzerspezifische Zulassungskontrollen durchgeführt, Darstellungselemente beinflußt oder auf Zeitereignisse reagiert werden. Eine detaillierte Übersicht über die Events, die auf dieser Ebene erzeugt und durch Methoden der Application Context Class verarbeitet werden können, ist im Referenzhandbuch für Kontextklassen (Online-Hilfe) enthalten. Application Context Classes können in ODABA2 als C++-Klassen implementiert werden.

4.3.3.3 Window Context

Windows sind Methoden einer Structure, die in einer Window Class implementiert werden. In diesem Sinne ist ein Window immer mit einer Structure verbunden. Der Window Context ermöglicht die Spezifikation speziellen Verhaltens, das sich im Zusammenhang eines Windows ergeben. Dazu kann die Berechnung abgeleiteter Werte genauso gehören, wie die Darstellung spezieller informationen in dem Window. Wesentlicher Bestandteil von Applikationen sind Menüs und Knöpfe, über die windowspezifische Aktionen angestoßen werden können. Derartige Menü- und Knopfaktionen können neben Window- und Document-Aktionen im Window Context definiert werden. Damit ist der Window Context Ausgangspunkt der Funktionalität eines Windows. Auch für Windows können benutzerspezifische Zulassungskontrollen durchgeführt und Darstellungselemente beinflußt werden. Eine detaillierte Übersicht über die Events, die auf dieser Ebene erzeugt und durch Methoden der Window Context Class verarbeitet werden können, ist im Referenzhandbuch für Kontextklassen (Online-Hilfe) enthalten.

Der Zusammenhang zwischen Window und Structure wird dadurch unterstützt, daß dem Window Context der Structure Context bekannt ist. Damit können in Methoden des Window Contexts Funktionen und OQL-Expressions des Structure-Contexts aufgerufen werden. Window Context Classes selbst können in ODABA2 jedoch nur als C++-Klassen implementiert werden.

4.3.3.4 Control Context

Controlls dienen der Anzeige von Property-Instanzen. Sie werden jedoch im Gegensatz zu Windows nicht als Methoden der Properties implementiert, sondern sind Bestandteil von Windoes. Dennoch ist ein Controll immer mit einer Property verbunden. Der Control Context ermöglicht die Spezifikation speziellen Verhaltens, das sich im Zusammenhang eines Controlls ergibt. Dazu gehört neben der Darstellung von Werten die Reaktion auf Eingaben in einem Control und die Reaktion auf definierte Accelerator Keys. Ebenso sind Eingabeprüfungen auf der Controll-Ebene sowie die Berechnung abgeleiteter Werte möglich. Eine detaillierte Übersicht über die Events, die auf dieser Ebene erzeugt und durch Methoden der Control Context Class verarbeitet werden können, ist im Referenzhandbuch für Kontextklassen (Online-Hilfe) enthalten.

Der Zusammenhang zwischen Control und Property wird dadurch unterstützt, daß dem Control Context der Property Context bekannt ist. Damit können in Methoden des Control Contexts Funktionen und OQL-Expressions des Property-Contexts aufgerufen werden. Control Context Classes selbst können in ODABA2 jedoch nur als C++-Klassen implementiert werden. Allerdings ist es über Action-Aufrufe möglich, andere Window- oder Documetaktionen sowie OQL-Expressions des Property-Contextes auszuführen.

4.3.4 Document Context Classes

Durch die Definition von Dokumenten im erweiterten ODABA2-Objektmodells ergibt sich ein weiterer Zusammenhang, dessen spezielles Verhalten durch Context Classes dargestellt werden kann. Da das Dukument in ODABA2 strukturell nicht

weiter gegliedert ist, wird in diesem Zusammenhang nur eine Context-Klasse, die Document Context Class, unterstützt.

4.3.4.1 Document Context

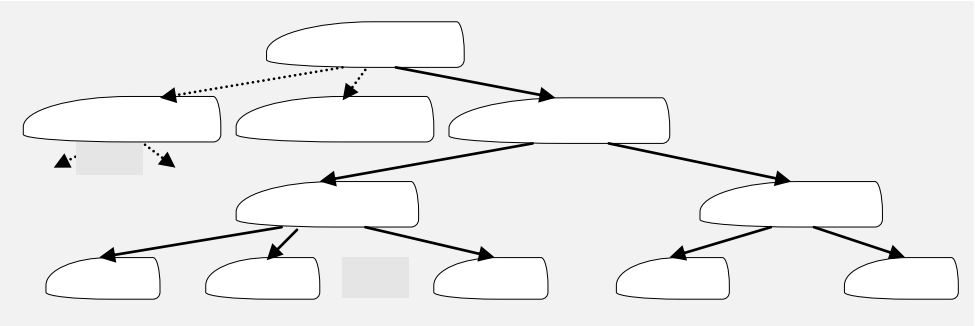
Der Ausgangspunkt für ein Dokument ist entweder eine Structure-Instanz oder ein Extent. Das Dokument ist in jedem Fall mit der Structure, die der Instanz oder dem Extent zugrunde liegt, verbunden. Im Dokumentkontext selbst ist es möglich, eine spezielle Instanz zur Darstellung auszuwählen, allgemeine Dokumentvariablen zu setzen sowie den Pfad für die Ausgabe des Dokumentes zu bestimmen. So können z.B. Dialoge gestartet werden, über die Parameter an das Dokument übergeben werden können. Eine detaillierte Übersicht über die Events, die auf dieser Ebene erzeugt und durch Methoden der Document Context Class verarbeitet werden können, ist im Referenzhandbuch für Kontextklassen (Online-Hilfe) enthalten.

Dokument Context Classes selbst können in ODABA2 als C++-Klassen implementiert werden. Allerdings ist es über Action-Aufrufe möglich, Window-Aktionen auszuführen.

4.4 Das erweiterte Objektmodell

Im Zusammenhang mit dem funktionalen Modell wird deutlich, daß eine Erweiterung des Objektmodells erforderlich ist, um diese komplexen Zusammenhänge darzustellen. Während man davon ausgehen kann, daß sich die Auffassungen zum elementaren Objektmodell zunehmend annähern werden, wird es vielfältige Wege geben, Methodische Zusammenhänge darzustellen. In ODABA2 selbst erlebt die Darstellung methodischer Zusammenhänge eine rasche Entwicklung. Aus diesem Grunde ist das ODABA2-Objektmodell offen, so daß beliebige erweiterte Objektmodelle zur Darstellung methodischer Zusammenhänge abgeleitet werden können. Damit werden auch methodische Darstellungen möglich, die sich aus ganz individuellen Erfordernissen ergeben. Das im folgenden skizzierte erweiterte Objektmodell ist also nur eine Variante, die in ODABA2 gewählt wurde, um methodische Zusammenhänge abzubilden.

In ODABA2 werden alle Methoden in Classes dargestellt, die Spezialisierungen von Structures sind. Während die Persistent Structures die Darstellung der Daten in der Datenbank definieren, werden die methodischen Zusammenhänge in den Classes abgebildet.



Dabei können zu einer Structure durchaus verschiedene Classes gebildet werden, genauso, wie Classes zu jeder beliebigen Structure gebildet werden können. So ist es also nicht nur möglich, Classes für Persisten Structures zu bilden, sondern auch Methoden auf der Basis von View-Structures zu implementieren. Context Classes sind aus Modellsicht ebenfalls einfache Classes, die auf einer definierten Structure basieren. Allerding gibt es für die Implementierung von Context Classes Einschränkungen, da sie nur als C++- oder als OQL-Classes implementiert werden können (es kann also kein Window definiert werden, mit dem Instanzen einer Context Class angezeigt werden).

4.4.1 Program Class

Program Classes sind Klassen, für die Methoden in einer Programmiersprache als Expressions oder Prozeduren implementiert werden. ODABA2 unterstützt dabei interpretiete und compilierte Methoden. Während interpretierte Methoden wesentlich flexibler und einfach zu implementieren sind, sind compilierte Methoden schneller und bieten meht Möglichkeiten. Derzeit ist es möglich, für eine

Structure sowohl eine OQL-Class als auch eine C++-Class zu erzeugen. Geplant ist außerdem die Unterstützung von JAVA.

4.4.1.1 C++-Classes

Über Methoden in C++-Classes ist der lesende und ändernde Zugriff auf Instanzen der Datenbank möglich. C++-Methoden sind zwar die effizienteste, aber auch die aufwendigste und anfälligste Variante der Implementierung von Methoden, da sich aus kleinen Änderungen oft vielfältige Übersetzungsanforderungen ergeben.

C++-Classes werden auf der Schemaebene als Ableitungen der SDB_Structure dargestellt. Details dazu können im Lexikon nechgelesen werden (→ **ODC_ImpClass**).

Die C++-Methoden werden im der Ressourcendatenbank gespeichert und erst zu Überstzungszeitpunkt als Class-Datei dem Compiler übergeben. Die Erzeugung der erforderlichen Include-Dateien erfolgt durch die ODABA2-Entwicklungsumgebung.

4.4.1.2 OQL-Classes

Über Methoden in OQL-Classes ist nur der lesende Zugriff auf Instanzen der Datenbank möglich. In begrenztem Umfang können über Initialisierungen im Variablenteil des Expressions auch Modifikationen einzelner Properties der Instanz vorgenommen werden. Da OQL-Methoden darüber hinaus C++-Funktionen der Context Classes aufrufen können, ist nicht immer gewährleistet, daß eine OQL-Funktion keine Änderungen ausführt. Die Implementierung von Methoden in OQL-Classes ist der einfachste Weg zur Implementierung von Methoden. Aufgrund der beschränkten Ausdrucksfähigkeit und der interpretierenden Verarbeitung sind sie jedoch nicht immer zur Darstellung der Methoden geeignet.

Methoden in OQL-Classes sind weitgehend unempfindlich auf Änderungen der zugrundeliegenden Structure-Definition und bedürfen keiner Übersetzung. Die OQL-Expressions werden im der Ressourcendatenbank gespeichert und erst bei

Aufruf interpretiert. OQL-Classes werden auf der Schemaebene als Ableitungen der SDB_Structure dargestellt. Details dazu können im Lexikon nechgelesen werden (→ **OQL_Class**).

4.4.2 Template Classes

In Template Classes können Methoden dargestellt werden, die einem speziellen Muster folgen (z.B. Dokumente, Tabellen oder Windows). Template-Methoden werden vor allem dann verwendet, wenn es um die Darstellung von Daten geht. Template-Methoden werden immer interpretierend verarbeitet. Dadurch ist es möglich, auch zur Lauzeit Änderungen schnell und einfach zu realisieren. ODABA2 unterstützt derzeit Template Classes für Windows und Dokumente.

4.4.2.1 Window Class

In einer Window Class können verschiedene Windows zur Anzeige einer Structure-Instanz definiert werden. Über Windows können Structure-Instanzen angezeigt oder bearbeitet werden. Über Methoden der Context Classes können darüberhinaus weitere Methoden aufgerufen werden, über die auch komplexe Aufgaben realisiert werden können. Die Implementierung von Methoden in Window Classes ist eine Einfache Möglichkeit zur Darstellung von Instanzen auf dem Bildschirm.

Methoden in Window Classes sind weitgehend unempfindlich auf Änderungen der zugrundeliegenden Structure-Definition und bedürfen keiner Übersetzung. Die Window-Definitionen werden im der Ressourcendatenbank gespeichert und erst bei Aufruf interpretiert. Window-Classes werden auf der Schemaebene als Ableitungen der SDB_Structure dargestellt. Details dazu können im Lexikon nechgelesen werden (→ **Win_Class**).

4.4.2.2 Document Class

In einer Dokument Class können Methoden zur Darstellung von Structure-Instanzen in einem Dokument definiert werden. Dabei definiert eine Methode der Document Class, ein Document Template, die Darstellung eines bestimmten Ausschnitts des Dukumentes. Dieser Ausschnitt kann ein einzelner Wert, eine Tabellenzeile, ein Kapitel oder auch ein vollständiges Dokument sein. Über Document Templates ist zwar nur die Anzeige der Daten im Dokument möglich. Da aus Dokument Templates jedoch OQL-Expressions bzw. C++-Funktionen des Structure Contextes aufgerufen werden können, ist auf diesem Wege auch die Modifikation der angezeigten Instanzen möglich. Über Methoden der Context Classes können darüberhinaus weitere Methoden aufgerufen werden, über die auch komplexe Aufgaben realisiert werden können. Die Implementierung von Methoden in Document Classes ist eine einfache Möglichkeit zur Darstellung von Instanzen in Dokumenten.

Methoden in Document Classes sind weitgehend unempfindlich auf Änderungen der zugrundeliegenden Structure-Definition und bedürfen keiner Übersetzung. Die Document-Definitionen werden im der Ressourcendatenbank gespeichert und erst bei Aufruf interpretiert. Document-Classes werden auf der Schemaebene als Ableitungen der SDB_Structure dargestellt. Details dazu können im Lexikon nechgelesen werden (→ **Doc_Class**).

Kapitel 5
Das Zustandsmodell

Das Zustandsmodell stellt die Verbindung zwischen Zuständen der Objektwelt und dem Verhalten der Objekte, zwischen den Structures und den Methoden her. Es beschreibt zum einen Zustände als Ausgangspunkt, Ursache für bestimmtes Verhalten. Zum anderen stellt es den Zusammenhang zwischen Zustandsübergängen und Reaktionen darauf dar.

Im ODABA2-Zustandsmodell basiert die Darstellung kausaler Zusammenhänge auf der Darstellung der im objektorientierten Sprachmodell beschriebenen Ursache/Wirkungs-Beziehung. Dabei werden Zustände oder Zustandsübergänge einer Instanz als Ursache aufgefaßt, wobei es von untergeordnetem Interesse ist, wie eine ursächlicher Zustand oder Zustandsübergang zustandegekommen ist. Als Ursache können Zustände oder Zustandsübergänge Reaktionen auslösen. Zustände und Zustandsübergänge werden durch Zustandsmengen dargestellt. Zustandsmengen können z.B. durch Aufzählung aller möglichen Zustände einer Zustandsmenge abgebildet werden. Zweckmäßiger hat sich jedoch die Darstellung durch logische Ausdrücke, durch **Conditions** erwiesen.

Kausale Zusammenhänge entstehen immer in einem bestimmten Umfeld. Mehr noch als das Verhalten ist die Darstellung kausaler Zusammenhänge somit an einen bestimmte Kontext gebunden. Das Verhalten in kausalen Zusammenhängen wird deshalb im Zustandsmodell durch kontextbezogene **Actions** abgebildet, die die Besonderheiten einer Methode in einem bestimmten Zusammenhang, im Context darstellen. Actions können durch direkte Aufforderungen oder durch besondere Zustandsübergänge, **Events**, ausgelöst werden. Conditions, Events und Actions sind die Grundbegriffe, auf denen die Darstellung aller kausalen Zusammenhänge beruht.

Im ODABA2-Zustandsmodell werden die im objektorientierten Sprachmodell eingeführten kausalen Beziehungen dargestellt, die als Bedingungen und Reaktionen beschrieben wurden. Aufforderungen werden in ODABA2 den

methodischen Zusammenhängen zugeordnet und im Zustandsmodell nicht weiter behandeln. Allerdings können Aufforderungen, die durch Actions implementiert werden, mit kontextbezogenen Bedingungen versehen werden und auf diese Weise bedingtes Verhalten im Zustandsmodell darstellen.

5.1 Conditions

Conditions beschreiben Zustandsmengen als Mengen potentieller Zustände. Während der konkrete Zustand einer Instanz durch ihren Wert bestimmt ist, sind potentielle Zustände Werte, die eine Instanz theoretisch annehmen könnte. Eine Condition definiert eine Zustandsmenge, indem alle potentiellen Zustände, für die die Condition den Wert wahr liefert, Element dieser Zustandsmenge sind.

Beispiel Wenn für das **Alter** von PERSONEN eine eingeschränkte Zustandsmenge $\mathbf{Z_0}$ mit

$$\mathbf{Z_0 = \{ z : z \in Z \text{ und } z >= 0 \text{ und } z <= 200 \}}$$

definiert ist, wobei **Z** die Menge aller möglichen potentiellen Zustände darstellt, dann ist **z = 22** ein Zustand der Zustandsmenge $\mathbf{Z_0}$, **z = 299** hingegen nicht. Das **Alter** von PERSONEN muß also zwischen 0 und 200 Jahren liegen.

Conditions werden als simple Actions (→ **CAU_simpleAction**) dargestellt, über die Methoden verschiedener Implementierungsklassen aufgerufen werden können. Somit kann eine Condition in gleicher Weise als OQL-Expression wie auch als C++-Funktion implementiert werden. Die aufgerufene Methode muß als Rückgabewert *wahr* oder *falsch* liefern. Als Actions sind Conditions nicht unbedingt an die Ausdrucksfähigkeit logischer Ausdrücke gebunden, sondern können bis hin zu Benutzereingaben verschiedenartige Einflußgrößen einbeziehen.

Da Zustände auf Instanzen bezogen sind, können Zustände für alle Instanzenarten definiert werden. Es können also Property-Conditions genauso wie Conditions für Structures oder Collections definiert werden.

5.1.1 Property Conditions

Property Conditions beschreiben Zustandsmengen für Property-Instanzen. Das schließt Zustandsmengen für Attributes genauso ein wie die Zustandsmengen von singulären References oder Collections. Der Einfachheit halber werden hier singuläre References als Collections aufgefaßt, die höchstens eine Instanz enthalten können, so daß alle References als Collections betrachtet werden können. In diesem Sinne werden Base Structures als eingebettete Structures und in diesem Sinne als Attributes betrachtet. ODABA2 unterscheidet im Rahmen von Property Conditions somit nur zwischen Attribute und Collection Condition.

Attribute Conditions

Zustände für Structured Attributes und Base Structures sind durch den Structure- oder Instanzenzustand ihrer Instanzen definiert. Zustandsmengen über diese Properties können also durch entsprechende Structure Conditions formuliert werden. Hier werden vorerst atomare Zustände, also Zustände von Attributes, deren Type ein Literal oder eine Enumeration ist, betrachtet. Der Zustand eines solchen Attributes entspricht dem Wert, den die Attribute-Instanz zu einem bestimmten Zeitpunkt in einer bestimmten Structure-Instanz besitzt.

> **Beispiel** Das **Alter** der PERSON *Anton Müller* ist ein atomarer Zustand, da es sich als eine natürliche Zahl, die selbst als unteilbar angesehen werden kann, ausdrücken läßt.
>
> Seine **Telefonnummer** kann als atomarer Zustand aufgefaßt werden, wenn sie als natürliche Zahl betrachtet wird, genauso gut kann sie aber auch als Folge von Ziffern betrachtet werden, womit sie keinen atomaren Zustand mehr repräsentiert.

Zustandsmengen für potentielle atomare Zustände sind die möglichen Werte für das entsprechende Attribute in einem bestimmten Zusammenhang. Dabei werden auf der Ebene der Properties keine Abhängigkeiten zu anderen Attributes berücksichtigt. Zustandsmengen für atomare Zustände sind durch technische Gegebenheiten begrenzt. So können z.B. ganzzahlige Werte nur im Rahmen von $-2^{31}+1$ bis $2^{31}-1$ dargestellt werden. Für Enumerations ist die Menge der zulässigen Zustände durch die Definition der Enumerators festgelegt. In diesem Sinne ist implizit oder explizit jedem atomaren Zustand entsprechend seinem Type eine

endliche Zustandsmenge **Z** zugeordnet, die die zulässigen Zustände für Attributes dieses Types definiert. Diese Zustandsmengen werden als **allgemeine Zustandsmengen** bezeichnet und geben den Rahmen vor, in dem die Speicherung von Daten überhaupt möglich ist. Die allgemeine Zustandsmengen können nun durch weitere Zustandsbedingungen eingeschränkt werden.

Beispiel Um eine Zustandsmenge für das zulässige **Alter** von PERSONEN festzulegen, kann eine Condition definiert werden:

```
{ self >= 0 && self < 200 }
```

Dabei bezeichnet **self** den konkreten Instanzenwert des Attributes. Wenn also das **Alter** einer konkreten PERSON die angegebene Bedingung erfüllt und zwischen 0 und 200 Jahren liegt, ist der konkrete Zustand ein Zustand der definierten Zustandsmenge.

Zustandsmengen für Attribute Arrays ergeben sich aus den Zustandsmengen der Array-Elemente. Ein potentieller Zustand **z** eines Attribute Arrays der Dimension **n** ist also durch

$$\mathbf{z = (z_1, ..., z_n)}$$

definiert, wobei $\mathbf{z_1, ..., z_n}$ atomare Zustände eines Attributelementes sind. Die Zustandsmenge **Z** ergibt sich dann aus der Teilmenge der Produktmenge der Zustandsmengen für die einzelnen Array-Elemente.

$$\mathbf{Z \subseteq Z_1 \times ... \times Z_n}$$

Zwischen den Array-Elementen eines Attributes können Abhängigkeiten definiert und als Condition dargestellt werden.

Beispiel Wenn für eine PERSON das **Alter_der_Kinder** in einem Array der Dimension 5 abgelegt werden soll, wobei die Altersangaben in der Größe aufsteigend sortiert sein sollen, kann eine Gültigkeitsbedingung für das Array als Condition zur Definition der Menge gültiger Zustände dieses Arrays definiert werden:

```
{ 0 <= self[0] and self[0] <= self[1]
               and self[1] <= self[2]
               and self[2] <= self[3]
               and self[3] <= self[4] and self[4] < 200}
```

self[i] bezeichnet dabei jeweils das **i+1**-ste Array-Element. Damit sind für **Alter_der_Kinder** nur Arrays mit aufsteigenden Altersangaben zwischen 0 und 200 Jahren zulässig.

Collection Conditions

Der Zustand einer Collection ist durch die Menge der referenzierten Structure-Instanzen festgelegt. Dabei hat der Zustand der Structure-Instanzen der Collection nur indirekten Einfluß auf den Zustand der Collection. Wenn eine Structure-Instanz der Collection ihren Zustand ändert, hat dies i. allg. keine Auswirkung auf den Zustand der Collection, solange die Instanz in der Collection verbleibt. Der Zustand einer Collection ändert sich nur, wenn Structure-Instanzen aus der Collection entfernt oder zu ihr hinzugefügt werden.

Beispiel Der Zustand der Collection **Kinder** einer PERSON ist durch die **Kinder**-Instanzen bestimmt. Er ändert sich, wenn **Kinder** hinzukommen oder aus der Collection entfernt werden. Ändert sich der Zustand der referenzierten **Kinder**-Instanzen, hat dies keinen Einfluß auf den Zustand der Collection. Um die Anzahl der **Kinder** für eine PERSON auf zehn zu begrenzen, könnte folgende Condition definiert werden:

```
{ GetCount() <= 10 }
```

GetCount() ist eine Methode für Collections, die die Anzahl der Instanzen der Collection liefert.

Conditions für die Definition von Zustandsmengen können durch die Kardinalität, die Anzahl der Structure-Instanzen in der Collection ausgedrückt werden. Zustandsmengen für Collections können jedoch auch dadurch definiert werden, daß eine Condition festlegt, welche Instanzen in der Collection gesammelt werden können.

Beispiel So kann eine Collection von *männlichen* PERSONEN definiert werden, indem eine Condition

```
{ geschlecht == männlich }
```

als Gültigkeitsbedingung der Collection angeben wird. In diesem Fall wird durch eine Structure Condition eine Menge von Instanzen festgelegt, die in dieser Collection gespeichert werden können.

Zustandsmengen für Collections beschreiben in diesem Sinne Zustandsmengen, die für alle Structure-Instanzen der Collection definiert sind. Im Gegensatz zu Zustandsmengen für Arrays können also für Collections keine definiert werden, die eine Abhängigkeit zwischen den einzelnen Instanzen ausdrücken Coditions (z.B. eine Ordnung oder ein Mindestabstand).

5.1.2 Structure Conditions

Der Zustand einer Structure-Instanz ergibt sich aus den Zuständen der einzelnen Property-Instanzen der Structure.

$$\mathbf{z = (z_1, ..., z_n)}$$

$\mathbf{z_i}$ wird in diesem Zusammenhang auch **Zustandskomponente** genannt. Die allgemeine Zustandsmenge **Z** für eine Structure ist dann die Produktmenge der Zustandsmengen ihrer Zustandskomponenten

$$\mathbf{Z = Z_1 \times ... \times}\ ,$$

wobei $\mathbf{Z_1}$, ... $\mathbf{Z_n}$ eingeschränkte oder allgemeine Zustandsmengen der Properties der Structure sind. Diese allgemeine Zustandsmenge kann nun durch eine Structure Condition eingeschränkt werden. Auf diese Weise können Abhängigkeiten zwischen den einzelnen Properties ausgedrückt werden.

Beispiel Wenn **Alter** und **Abschluß** Attributes von PERSON sind, können wir davon ausgehen, daß eine PERSON mit *Hochschulabschluß* älter als 20 Jahre sein muß. Eine Gültigkeitsbedingung für PERSONEN wäre dann:

```
{ !(Abschluß == Hochschulabschluß && Alter < 20) }
```

Zwischen den Zustandskomponenten **Abschluß** und **Alter** besteht also eine Abhängigkeit, die den Zustand

{*Ernesto Müller, Hochschulabschluß, 15 Jahre*}

als zulässigen Zustand aus einer so definierten Zustandsmenge ausschließt, obwohl jede Property-Instanz für sich mit einem gültigen Wert belegt ist.

Der Zustand einer Structure-Instanz ändert sich, wenn sich der Zustand einer ihrer Property-Instanzen ändert, die Base Structure-Instanzen eingeschlossen. Die Zustandsänderung einer Structure-Instanz kann also weitreichende Folgen haben, da sich aus der Zustandsveränderung einer Structure-Instanz automatisch Zustandsveränderungen für alle abgeleiteten Instanzen ergeben. Damit abgeleitete Instanzen auf Zustandsänderungen ihrer Basisinstanzen reagieren können, muß die Basisinstanz alle ihre abgeleiteten Instanzen kennen. Werden Base Structures als Relationships und somit als freie Instanzen dargestellt, bedeutet das, daß die inversen References für die abgeleitete Structures definiert sein müssen, damit ODABA2 die logische Konsistenz bezüglich der definierten Gültigkeitsbedingungen sichern kann.

5.1.3 Gültigkeitsbedingungen

Obwohl Gültigkeitsbedingungen ein Spezialfall der Verwendung von Zustandsmengen sind, sollen sie hier noch einmal gesondert behandelt werden, da sie einen typischen und häufig verwendeten Anwendungsfall für Conditions darstellen. Gültigkeitsbedingungen definieren eine eingeschränkte Zustandsmenge für eine Property- oder Structure-Instanz, die die Menge der zulässigen Instanzen beschreibt. ODABA2 sichert in diesem Sinne die logische Konsistenz der Datenbank, als daß keine Instanzen gespeichert werden, die nicht den angegebenen Gültigkeitsbedingungen entsprechen.

Für den Fall, daß Gültigkeitsbedingungen geändert werden, indem weitere Einschränkungen getroffen werden, können nachträglich logische Inkonsistenzen für bereits gespeicherte Instanzen entstehen. Diese logische Inkonsistenz der Datenbank kann dann nur durch einen Reorganisationsprozeß oder durch spezielle Anwenderprogramme beseitigt werden.

Die Frage nach der Gültigkeit von Zuständen schließt sowohl die Frage nach gültigen Zuständen als auch die nach gültigen Zustandsübergängen ein. In ODABA2 gibte es verschiedene Möglichkeiten zur Definition gültiger Zustände und gültiger Zustandsübergänge.

Gültige Zustände können als Property, Collection oder Structure Conditions definert werden. Structure Conditions definieren dabei kontextfreie Bedingungen, während durch Property und Collection Conditions kontextbezogene Bedingungen definiert werden.

Vielfach läßt sich ein Zustand jedoch nicht allgemein als gültig oder ungültig erklären. Auch der Context, in dem ein Zustand betrachtet wird, reicht oft nicht aus, da er keine Information über die Entwicklung der Instanz, über ihre Vergangenheit enthält. Um Conditions für Zustandsübergänge definieren zu können, müssen alter und neuer Zustand der Instanz im Expression verfügbar sein. Das ist immer dann der Fall, wenn eine Zustandsveränderung der Instanz stattfindet. Dabei kann die neue Instanz über **self**, die alte hingegen über **before** angesprochen werden.

Beispiel Wenn für PERSONEN ein Zustand **Nachkommen** dargestellt wird, der mitteilt, ob eine PERSON *Kinder*, *Enkel*, *Urenkel* hat oder *kinderlos* ist, kann eine Person zwar jeden dieser Zustände annehmen. Es ist jedoch schwer vorstellbar, daß *Jacqueline Müller*, bisher kinderlos, plötzlich Großmutter wird. Der Zustandsübergang *kinderlos* → *Enkel* ist also ein verbotener Zustandsübergang.

Zustandsübergänge werden in ODABA2 durch Events definiert. In diesem Sinne ist auch die Reaktion auf ungültige Zustandsübergänge im Rahmen der Events und Reactions angesiedelt.

5.2 Events

Ein Event definiert einen bedeutenden Zustandsübergang, ein Ereignis. Zustandsübergänge sind Zeitpunkte, in denen eine Instanz ihren Zustand, ihren Wert ändert. Potentielle Zustandsübergänge lassen sich durch zwei Zustandsmengen, zwei Conditions darstellen, die die Zustände vor dem Zustandsübergang und die nach dem Zustandsübergang festlegen. Events beschreiben Zustandsübergänge, die zur Auslösung einer Action führen können, nehmen jedoch nicht auf die auszulösenden Actions, die sehr vielfältig sein können, bezug. Insofern sind Events potentielle Ereignisse, die erst beim konkreten Eintreten eines Events mit einer Action verbunden werden.

5.2.1 Definition von Events

Events werden als ausgewählte Zustandsübergänge dargestellt, die durch eine Menge von Ausgangs- und Endzuständen definiert sind. Ein Event tritt ein, wenn die Instanz vor einer Änderung einen Instanzenzustand aus der Menge der Ausgangszustände besaß und nach der Änderung ein Zustand der Menge der Endzustände angenommen hat. Wenn die Zustandsmenge $\mathbf{Z_A}$ eine Menge von Anfangszuständen vor dem Zustandsübergang und die Zustandsmenge $\mathbf{Z_E}$ eine Menge von Endzuständen nach dem Zustandsübergang beschreibt, dann definiert $\mathbf{Z_T}$ mit

$$\mathbf{Z_T} = \{ \mathbf{(z_a, z_e)} : \mathbf{z_a} \in \mathbf{Z_A} \text{ und } \mathbf{z_e} \in \mathbf{Z_E})$$

als Menge von potentiellen Zustandsübergängen ein Event. Events bezeichnen also ebenfalls potentielle Zustandsübergänge und keine konkreten. Für eine Instanz tritt ein Event ein und wird konkret, wenn die Instanz ihren Zustand von $\mathbf{z_a}$ nach $\mathbf{z_e}$ ändert, wobei $\mathbf{(z_a, z_e) \in Z_T}$, also aus der Menge der potentiellen Zustandsübergänge dieses Events ist.

Beispiel Wenn sich das **Alter** einer Person von 99 auf 100 ändert, ist das ein Ereignis, das in besonderer Weise gewürdigt werden soll. Dieses kann durch zwei Zustandsmengen, zwei Conditions dargestellt werden:

```
{ self == 99  }                    // Anfangszustand
{ self == 100 }                    // Endzustand
```

Hier löst nur der Übergang von 99 auf 100 Jahre ein Event aus. Eine Änderung des Alters von 20 auf 21 Jahre würde nicht zum Ereignis führen.

Spezielle Events sind solche, für die die Menge der Anfangszustände oder die Menge der Endzustände gleich der allgemeinen oder eingeschränkten Zustandsmenge **Z** einer Instanz ist. Ist $\mathbf{Z_A = Z}$, ist nur noch der Endzustand $\mathbf{Z_E}$ von Interesse, um einen konkreten Zustandsübergang einem Event zuzuordnen. Auf diese Weise können ebenfalls allgemeine Gültigkeitsbedingungen definiert werden, wobei die Möglichkeit der Reaktion auf Verletzung der Gültigkeitsbedingungen im Rahmen von Events umfangreicher ist.

Ist $\mathbf{Z_E = Z}$, signalisiert jeder Zustandsübergang eines Anfangszustandes aus $\mathbf{Z_A}$ ein Event. Ist ein Event hingegen durch $\mathbf{Z_A = Z}$ und $\mathbf{Z_E = Z}$ definiert, wird jeder konkrete Zustandsübergang, jede Veränderung einer Instanz als Event aufgefaßt. Die letzteren Events werden als **modify** Events bezeichnet, da sie eine beliebige Veränderung einer Instanz anzeigen. Da ein Event durch zwei Zustandsmengen dargestellt wird, kann es durch zwei Conditions abgebildet werden (→ **CAU_Event**).

- **Pre-Condition**

  Die Pre-Condition legt die Menge der Anfangszustände für das Event fest. Für eine konkrete Instanz liegt ein gültiger Anfangszustand vor, wenn die Pre-Condition den Wert *wahr* liefert.

- **Post-Condition**

  Die Post-Condition legt die Menge der Endzustände für das Event fest. Für eine konkrete Instanz liegt ein gültiger Endzustand vor, wenn die Post-Condition den Wert *wahr* liefert.

Ein Event tritt also ein, wenn für eine Instanz vor einer Änderung die Pre-Condition und nach einer Änderung die Post-Condition *wahr* ist. Events können für Properties oder Structures definiert werden. Das Eintreten eines Events bezüglich einer Instanz wird festgestellt, indem beim Aktivieren einer Instanz geprüft wird, ob die Pre-Condition erfüllt ist. Ist dies der Fall, wird bei jeder Änderung die Post-Condition geprüft. Ist sie ebenfalls *wahr*, wird das Ereignis signalisiert. Natürlich wird bei Änderungen auch die Pre-Condition neu bestimmt, so daß bei der nächsten Änderung wieder vom aktuellen Zustand der Instanz ausgegangen wird. Ist keine Pre-Condition angegeben (immer wahr), wird bei jeder Änderung einer Instanz die Post-Condition geprüft.

Änderungen einer Instanz werden in einer C++-Umgebung i.allg. durch die **Modify()**-Funktion bekanntgegeben, da ODABA2 keine Kenntnis von Änderungen an Feldern im Rahmen eines C++-Programms erlangt. Nur in OQL-Expressions oder bei der Verwendung von DBFields zur Modifikation von Properties können Events auf der Ebene von Property-Instanzen sofort erzeugt werden.

Da Events immer an einen Zustandsübergang einer Instanz gebunden sind, werden sie auf der Schemaebene den Structures und den Properties zugeordnet, d.h. Events werden als Eigenschaften der Structure- bzw. Property-Definition aufgefaßt. Die Methoden zur Berechnung der Conditions werden in den Context Classes definiert, die für die Properties oder Structures gebildet werden können. Können Events nur durch Zustandsübergänge mehrere Instanzen definiert werden (kooperative Events) besteht die einfachste Möglichkeit darin, eine entsprechende Association der betroffenen Structures zu bilden und die Events auf diesen funktionalen Zusammenhang bezogen zu definieren.

5.2.2 Events auf verschiedenen Ebenen

Genauso wie das Verhalten können auch Events auf verschiedenen Ebenen spezifiziert werden. ODABA2 kennt Events auf der System- und auf der Context-Ebene eingehen. Auf der Problemebene ergeben sich zwar ebenfalls vielfältige Möglichkeiten der Definition von Events. Diese könne wir jedoch auch als Events in einem allgemeinen Structure Context auffassen. Die Beschränkung auf die System- und Context-Ebene beruht vor allem darauf, daß es derzeit nur auf diesen Ebenen möglich ist, auf Events zu reagieren.

Events der System-Ebene

Auf System-Ebene können Zustandsmengen nur allgemein ohne Bezug auf die Properties definiert werden. Damit können als Zustandsübergänge auf dieser Ebene nur Events wie z.B. das **modify** Event, das eine beliebige Veränderung anzeigt, definiert werden. Da auf der System-Ebene die Bedingungen für die Events vordefiniert sind, entfällt die Definition von Pre- und Post-Conditions. Die Events der Systemebene sind im Anhang beschrieben (→ **DB_Event**).

Allen System Events ist gemeinsam, daß sie nach dem Zustandsübergang signalisiert werden, d.h. der Zustandsübergang kann nicht verhindert werden. Die zu löschende Instanz ist also bereits gelöscht oder aus der Collection entfernt worden, genauso, wie das Hinzufügen oder Lesen schon stattgefunden hat und nur das bereits eingetretene Ereignis signalisiert wird. Um die Action selbst (also das Hinzufügen oder Löschen) zu verhindern, können entsprechende Standard-Actions überladen und mit Bedingungen versehen werden. Das Event ist auf jeden Fall genauso ungeeignet, eine Action zu verhindern, wie das Ereignis die Ursachen nicht zurücknehmen kann, die zum Auslösen des Ereignisses geführt haben, so wünschenswert dies manchmal auch scheinen mag.

Events der Context-Ebene

Events auf der Context-Ebene sind keine vordefinierten Events, wie die Events der System-Ebene, sondern werden anwendungsbezogen als Zustandsübergang durch Pre- und Post-Condition definiert. Durch die kontextbezogene Definition von Events wird es außerdem möglich, Ereignisse nach Zusammenhängen zu differenzieren. Auf diese Weise können relevante Zustandsübergänge in speziellen Zusammenhängen als Events definiert werden, während sie in anderen Zusammenhängen kaum ein Rolle spielen. So ist zum Beispiel ein Geburtstag nur im Zusammenhang mit Freunden oder Verwandten von Interesse, während Unbekannte keinen Anlaß sehen, auf diesen Zustandsübergang zu reagieren. In diesem Sinne ist es sinnvoll, Events nur in dem Kontext zu erzeugen, in dem sie eine Rolle spielen.

Auf der Context-Ebene ergeben sich vielfältige Möglichkeiten der Definition von Events aus inhaltlicher Sicht. Das betrifft hauptsächlich Zustände von Structure- und Attribute-Instanzen. Die Darstellung der Zustandsmengen für die Events erfolgt auch hier durch Conditions, die als Methoden der entsprechenden Context Class implementiert werden können. Da auf der Context-Ebene beliebig viele Events definiert werden können, werden sie mit einem Namen versehen, auf den bei der Reaktion auf das Event Bezug genommen werden kann.

5.3 Actions

Ist die Ausführung eines Verhaltens an einen kausalen Zusammenhang gebunden, wird es im Zustandsmodell durch **Actions** abgebildet, die die Besonderheiten einer Methode in einem bestimmten Zusammenhang, im Context darstellen. Während die Methode eine potentielle Fähigkeit ausdrückt, stellt die Action ein Verhalten in einem kausalen Zusammenhang dar. Actions können durch direkte Aufforderungen in Methoden oder durch **Events** ausgelöst werden. Dabei kann es geschehen, daß eine Methode, die potentiell ausgeführt werden könnte, in einem bestimmten Zusammenhang nicht ausführbar ist. Im Gegensatz zur Methode hat die Action die Eigenschaft, die Zulässigkeit der Methode in einem bestimmten Umfeld zu prüfen und kontextbezogen Folgen der Methode nach Ausführung darzustellen.

Im folgenden sollen die wesentlichen Zusammenhänge und die Verwendung von Actions dargestellt werden. Eine detaillierte Beschreibung ist im Lexikon unter →**CAU_Action** zu finden.

Im Gegensatz zu den Methoden, die auf der Problemebene im Rahmen einer Structure definiert werden, bilden Actions das Verhalten in einem bestimmten Zusammenhang, im Context ab. Actions sind also mit einer Methode, einem Action Handler verbunden, der im entsprechenden Structure oder Property Context definiert ist. Durch Pre- und Post-Handler ist nicht nur eine Vor- und Nachbehandlung der Action möglich, sondern auch das Prüfen der Zulässigkeit der Action. So kann über den Pre-Handler eine Prüfung erfolgen und die Ausführung der Action verhindert werden. Pre- und Post-Handler sind ebenfalls Methoden des entsprechenden Contexts.

Beispiel 6.15 Wenn der **Chef** eines TEAMS das **Gehalt** eines MITARBEITERS verändern will, ist er dazu nur in den Grenzen eines **Fonds** berechtigt, der ihm zur Verfügung gestellt wurde. In diesem Zusammenhang wird vor der Erhöhung geprüft, ob die Action **neuesGehalt** ausgeführt werden kann.

```
structure Mitarbeiter {
   attribute( Name    { type STRING; size 30 }
              Vorname { type STRING; size 30 }
```

```
                Gehalt  { type INT; size 8; precision 2 }
                Fonds   { type INT; size 9; precision 2 }
    )
       relationship(
                Chef    { type Mitarbeiter; inverse Team }
                Team    { type Mitarbeiter; inverse Chef;
                          action neuesGehalt
                          { pre_handler CheckFonds;
                            act_handler
                             Mitarbeiter.Gehaltsveränderung
    }  }            )
    }
```

Da die Action **neuesGehalt** nur im Context der Collection **Team** definiert ist und das **Team** nur im Zusammenhang mit einem **Chef** existiert, kann nur über den **Chef** die **Gehaltsveränderung** für einen MITARBEITER seines **Teams** erfolgen. Es kann also kein MITARBEITER sein eigenes **Gehalt** erhöhen, solange der **Chef** nicht als MITARBEITER seines **Teams** definiert wird. **Gehaltserhöhung** kann als Methode der Context Class MITARBEITER implementiert werden. **CheckFond** ist eine Methode, die im Collection Context für **Team** oder ebenfalls im Structure Context für MITARBEITER implementiert wird.

Durch die kontextbezogene Definition von Actions kann nicht nur die Ausführung einer Action an einen Context gebunden werden, wie im obigen Beispiel, sondern es können auch spezielle Behandlungen in diesem Context ausgeführt werden.

5.3.1 Der Pre-Handler

Der Pre-Handler einer Action kann zum einen eine Condition definieren, die die erforderlichen Ausgangszustände für die Action definiert. Auf diese Weise können bedingte Ac-tions definiert werden, die darstellen, unter welchen Umständen eine Action ausgeführt werden kann. Zum anderen kann der Pre-Handler aber auch Maßnahmen zur Vorbereitung der Action einleiten und geht somit über die Möglichkeiten einer Condition hinaus.

Beispiel Beim Abheben eines Betrages von einem Konto wird am Geldautomat der Geheimcode abgefragt, direkt am Schalter dagegen nicht. Die Abfrage des Geheimcodes ist also vom Umfeld, vom Context abhängig, in dem das Geld abgehoben wird. Bei der Darstellung dieses Sachverhaltes kann jedoch die gleiche Methode verwendet werden, wenn das Verhalten als Action definiert und im jeweiligen Context mit einem Pre-Handler verbunden wird.

Durch den Pre-Handler können Zulässigkeitsprüfungen und vorbereitende Maßnahmen für das Ausführen der Action-Methode vorgenommen werden. In Abhängigkeit vom Rückgabewert der Methode des Pre-Handlers kann die Ausführung der Action-Methode unterbunden werden. Da der Pre-Handler als Condition nicht unbedingt ein logischer Ausdruck sein muß, können über den Pre-Handler auch Schutzfunktionen wie Berechtigungskontrollen oder Passwort-Abfragen realisiert werden.

5.3.2 Der Action Handler

Die durch den Action Handler definierte Methode realisiert das eigentliche Verhalten der Action. Indem alle kontextspezifischen Aktivitäten in den Pre- und Post-Handler verlagert werden, wird die Methode von dem Wissen um die speziellen Zusammenhänge befreit und somit universell einsetzbar. Durch die Trennung kontextspezifischer und problemrelevanter Funktionalität wird eine wesentlich höhere Portabilität der Methoden erreicht. Da die Methoden zur Darstellung des Verhaltens nun weniger an den konkreten Zusammenhang gebunden sind, können sie letztlich überall dort eingesetzt werden, wo dieses Verhalten an sich gefragt ist.

Beispiel Das Entfernen einer Instanz aus einer Collection ist sicher ein ganz allgemeiner Vorgang. Deshalb wird er auch für alle Collections durch ein und dieselbe Methode realisiert. Dennoch gibt es in bestimmten Zusammenhängen Anforderungen, das Löschen einer Instanz nur unter bestimmten Bedingungen auszuführen. Indem die entsprechende Action im gegebenen Context überladen wird, wird zwar die gleiche Methode zum Löschen der Instanz verwendet, jedoch ist es durch den Pre-Handler möglich, das Löschen zu verhindern, wenn die Bedingungen nicht erfüllt sind.

Action Handler werden als Methoden der Context Class implementiert, die mit dem Context verbunden ist. Dabei sind verschiedene Implementierungsarten möglich. So kann die Methode einer Action als Programmfunktion, aber auch als Window oder Document Template implementiert werden. Dadurch wird es möglich, Actions direkt mit einem Dokument oder einer Eingabemaske zu verbinden. Die Art der Implementierung der Methode hängt von dem Umfeld ab, in dem die Action ausgeführt werden soll, da sie nur ausführbar ist, wenn die Umgebung den gewählten Action-Typ unterstützt.

5.3.3 Post-Handler

Post-Handler ermöglichen die kontextspezifische Nachbehandlung einer Action. Die Nachbehandlung besteht in der Ausführung von Maßnahmen, die nicht unmittelbar mit dem Verhalten verbunden, sondern nur in einem bestimmten Umfeld erforderlich sind. Gerade im Zusammenhang mit graphischen Oberflächen ist es oft der Fall, daß durch das Verhalten vorgenommene Zustandsveränderungen auch in der Maske auf dem Bildschirm angezeigt werden sollen. Dies ist jedoch nicht die Aufgabe der Methode, die damit an die graphische Oberfläche gebunden wäre, sondern Aufgabe des Post-Handlers, der im Context der Oberfläche definiert werden kann. Genauso gibt es auch im ODM kontextbezogene Maßnahmen, die als Nachbehandlung spezifiziert werden können. Im ODM können Post-Handler verwendet werden, um kontextspezifische Nachbehandlungen einer Aktion auszuführen.

Beispiel Infolge einer **Gehaltsveränderung** im Rahmen einer ABTEILUNG muß der **Fonds** der ABTEILUNG aktualisiert werden. Da dies nicht erforderlich ist, wenn das **Gehalt** eines MITARBEITERS der GESCHÄFTSLEITUNG verändert wird, der keiner ABTEILUNG angehört, ist diese Nachbehandlung kontextbezogen, d.h. für **Mitarbeiter** der GESCHÄFTSLEITUNG erfolgen andere Maßnahmen als für die **Mitarbeiter** einer ABTEILUNG.

```
structure Mitarbeiter {  ...
   relationship Team { type Mitarbeiter; inverse Chef;
                       action neuesGehalt
                       { pre_handler CheckFonds;
                         act_handler Gehaltsveränderung;
                         post_handler SetupFonds } }
}
```

Im Context **Team** werden **Gehaltsveränderungen** nun durch den Post-Handler behandelt, indem der **Lohnfonds** der ABTEILUNG aktualisiert wird. Die Methode **Gehaltsveränderung** hingegen ist nicht vom konkreten Zusammenhang abhängig und kann an beliebiger Stelle verwendet werden.

Post-Handler werden wie Pre-Handler als Methoden der entsprechenden Context Class implementiert. Meistens werden Pre- und Post-Handler als Programmfunktionen oder OQL-Expressions implementiert. Dennoch ist es prinzipiell möglich, auch für Post-Handler Window oder Document Templates zu implementieren.

5.3.4 Actions auf der System-Ebene

Auf der System-Ebene sind verschiedene Actions vordefiniert, die die Ausführung von Standard-Actions ermöglichen. Bei der Beschreibung der Events haben wir gesehen, daß Events immer nur einen vollzogenen Zustandsübergang anzeigen. Events bieten also keine Möglichkeit, bestimmte Zustandsveränderungen zu verhindern, wie sie gerade auf der Systemebene z.B. beim Hinzufügen und Löschen erfolgen. Deshalb werden typische Methoden der System Classes als Actions definiert, so daß sie im jeweiligen Context überladen werden können. Durch kontextbezogene Pre-Handler können dann spezielle Prüfungen vorgenommen und die Ausführung der Methode ggf. verhindert werden. Wird z.B. im Collection Context eine Instanz hinzugefügt, erfolgt dies intern nicht durch einen entsprechenden Methodenaufruf, sondern durch das Auslösen einer Action. Dadurch ist es möglich, die Action im Context zu überladen und mit kontextspezifischer Vor- und Nachbehandlung auszustatten.

Auf ähnliche Weise wie die Pre-Condition für Events, kann auch der Pre-Handler für eine Action durch eine Condition eine Zustandsmenge definieren, die die Ausgangszustände beschreibt, die zur Ausführung der Methode vorliegen müssen. Da ein Event jedoch einen vollzogenen Zustandsübergang anzeigt, können Events nicht verhindert werden. Ein **Insert** Event würde also immer das vollzogene Hinzufügen einer Instanz zur Collection signalisieren, nicht aber die bevorstehende Ausführung der Methode, wie der Pre-Handler der **Insert** Action.

Eine detailierte Beschreibung der System Actions, die in den verschiedenen Contexts überladen werden können, ist im Reference-Handbuch (Online-Hilfe) enthalten.

5.4 Reactions

Reactions bilden die Verbindung zwischen einem Event und der durch das Event ausgelösten Action ab. Im Gegensatz zum direkten Aufruf einer Action oder Methode realisiert die Reaction keine gezielte Aufforderung, d.h. die Event-auslösende Instanz weiß nichts von der oder den ausgelösten Reactions. Da Events von mehreren Instanzen wahrgenommen werden können, kann ein Event durchaus zum Auslösen mehrerer Actions führen. Es ist eine Sache der anderen Structure-Instanzen, Events zu bemerken und darauf zu reagieren, indem sie ein bestimmtes Ereignis beobachten. Events können von beliebigen Instanzen beobachtet oder an andere Instanzen signalisiert werden. Da die Event-auslösende Instanz nicht von den Reactions anderer Instanzen weiß, kann sie diese i.allg. auch nicht auswerten. Es ist ihr sozusagen egal, was andere mit dem Event machen.

Natürlich ist es kaum vertretbar, daß jede Instanz alle anderen ständig beobachtet, um eventuell auf Events reagieren zu können. Es macht auch umgekehrt wenig Sinn, ein Event allen Instanzen der Datenbank oder der Applikation zu signalisieren. Die Anwendung wäre dann nur noch mit sich selbst beschäftigt. Deshalb werden eingetretene Events nur von bestimmten Instanzen beobachtet und an diese weitergeleitet. Welche Instanzen dabei von einer anderen Instanz beobachtet werden, ist Bestandteil der Definition kausaler Beziehungen im Zustandsmodell und kann auf verschiedene Weise dargestellt werden.

Da Events als Zustandsübergänge von Instanzen konkret sind, ereignen sie sich immer in einem bestimmten Zusammenhang, im Context, in dem die Instanz erscheint. Da eine Instanz in mehreren Zusammenhängen auftreten kann, kann auch ein Event in mehreren Zusammenhängen eine Rolle spielen. ODABA2 geht deshalb davon aus, daß die Beobachtung und Reaktion auf Events in einem oder mehreren Contexts dargestellt werden. Aus der Sicht der Datenbank gibt es verschiedene Varianten, auf Events, die von einer Datenbankinstanz ausgelöst wurden, zu reagieren:

- **im eigenen Context**

  Da jede Instanz einen eigenen Context besitzt, liegt es nahe, auf eingetretene Events zuerst im eigenen Kontext zu reagieren.

- **im Context verwandter Instanzen**

  Verwandte Instanzen sind Instanzen, zwischen denen ein Instanzenpfad definiert werden kann. Verwandte Instanzen stehen also über eine oder mehrere Stufen in einer direkten oder indirekten strukturellen Beziehung zueinander. Verwandte Instanzen sind also die unmittelbar oder über mehrere Stufen referenzierten Instanzen von References und Relationships.

- **im übergeordneten Context**

  Da die Reaction auf Events in verwandten Instanzen an die Existenz von References gebunden ist, die für lokale References nur auf die nachgeordneten Instanzen existieren, sind spezielle Mechanismen erforderlich, um auf Events im übergeordneten Context zu reagieren.

- **im Context entfernter Instanzen**

  Es ist möglich, daß sich entfernte Instanzen, also nicht verwandte Structure-Instanzen, für bestimmte Events interessieren und somit einen Beobachterstatus erhalten. In diesem Fall wird der kausaler Zusammenhang zwischen den entfernten Instanzen direkt definiert, ohne daß es einen strukturellen Zusammenhang gibt.

Da die Contexts die gespeicherten Zusammenhänge beschreiben, müssen reagierende Instanzen nicht unbedingt aktiv, also eingelesen worden sein, wenn das Event eintritt. Prinzipiell können also sowohl inaktive als auch aktive Instanzen auf Ereignisse Reagieren. In vielen Fällen sollen aber nur aktive Instanzen auf ein Event reagieren.

Beispiel Wenn im Rahmen einer **Gehaltsveränderung** die **Lohnsumme** der ABTEILUNG aktualisiert werden soll, muß das unabhängig davon geschehen, ob die ABTEILUNGS-Instanz aktiv ist oder nicht. Ist jedoch **Lohnsumme** eine transiente Property, die erst beim Einlesen der ABTEILUNGS-Instanz berechnet wird, ist eine Reaktion auf die Veränderung des **Gehalts** eines MITARBEITERS nur erforderlich, wenn die ABTEILUNGS-Instanz bzw. ihr Structure Context aktiv ist.

Wenn die Instanzen, die andere Instanzen beobachten, bekannt sind, können eingetretene Events zielgerichtet an die beobachtenden Instanzen signalisiert werden. Ist dies nicht der Fall, werden die Events allen Instanzen signalisiert, die möglicherweise auf das Event reagieren wollen. Dabei muß es sich nicht nur um Structure-Instanzen handeln. Es können genausogut Collection- oder Property-Instanzen auf Events reagieren. Da die ausgelöste Action wieder über Pre- und Post-Handler verfügt, kann das Ausführen der Action-Methode kontextbezogen geprüft und ggf. unterbunden werden.

Im Context der beobachtenden Instanzen wird die Reaction auf ein Event durch die Verbindung eines Events mit einer Action dargestellt, die bei Eintreten eines Events ausgeführt wird. Ist die reagierende Instanz eine Collection-Instanz, bedeutet das i.allg., daß alle Instanzen der Collection auf das Event reagieren sollen. Für die Darstellung dieses kausalen Zusammenhangs zwischen Instanzen gibt es je nach Art der Beziehung unterschiedliche Möglichkeiten, die im folgenden beschrieben werden.

5.4.1 Reactions der eigenen Instanz

Es scheint erst einmal etwas eigenartig, daß eine Instanz in ihrem eigenen Context auf ein Event, also eine bestimmte Veränderung ihrer selbst, reagieren soll. Dies erweist sich jedoch immer dann als vorteilhaft, wenn die Zustandsübergänge nicht im eigenen Context verursacht wurden, wie es häufig bei System Events der Fall ist. Alle System Events werden zuerst dem eigenen Context signalisiert. So wird das Ändern einer Structure-Instanz dem entsprechenden Structure Context, das Hinzufügen oder Löschen einer Instanz aus einer Collection dem entsprechenden Collection Context mitgeteilt. Es können jedoch auch spezielle Zustandsübergänge einer Instanz als Event definiert und dem Context mitgeteilt werden.

> **Beispiel** Wenn für die MITARBEITER-Structure ein transientes Attribute **Durchschnitt** definiert wurde, in dem das durchschnittliche Jahresgehalt ausgewiesen werden soll, ist es am einfachsten, beim Einlesen der MITARBEITER-Instanz das **read** Event im Structure Context mit einer Action zu verbinden, so daß der Wert für **Durchschnitt** aus den monatlichen Einkommen berechnet werden kann.

Ohne die Verwendung von Reactions auf das Event müßte bei jedem Einlesen in verschiedenen Zusammenhängen darauf geachtet werden, daß die transienten

Attributes korrekt berechnet werden. Durch die Reaction wird dieses Problem ein für allemal gelöst. Allerdings gibt es gerade für die Bearbeitung von System Events auch einfachere Verfahren im funktionalen Modell. Indem das System Event in der System Class mit einer virtuellen Funktion assoziiert wird, kann sie im Structure Context überladen werden, der von der Structure System Class abgeleitet ist. Nun wird die Methode direkt gerufen, wenn das entsprechende System Event ausgelöst wird. Da Context Classes für alle Context-Arten gebildet werden können, gilt dieses Prinzip in gleicher Weise auch für Collection-, Property- und Real Object Contexts.

5.4.2 Reactions verwandter Instanzen

Inbesondere kontextbezogene Abhängigkeiten erfordern immer wieder, daß **modify** oder ähnliche Events verwandten Instanzen signalisiert werden. So kann also auch in verwandten Instanzen auf Events reagiert werden.

> **Beispiel 6.23** Um zu sichern, daß die **Lohnsumme** der ABTEILUNG auch bei Änderung des Gehalts eines MITARBEITERS den richtigen Wert enthält, muß die ABTEILUNGS-Instanz auf jede Änderung einer MITARBEITER-Instanz reagieren.

Es ist ein Unterschied, ob eine Reaction nur für aktive Instanzen oder für alle Instanzen des Contexts erfolgen soll. Während auf ein **read** Event i.allg. nur durch aktive Instanzen reagiert wird, ist die Aktualisierung des **Lohnfonds** der ABTEILUNG in jedem Falle erforderlich, auch wenn die ABTEILUNGS-Instanz nicht aktiv ist. Die Unterstützung verwandter Contexts ist für inaktive Contexts nur möglich, wenn die erforderlichen Verbindungen über References oder Relationships hergestellt wurden, d.h. es muß einen Property-Pfad von der reagierenden Instanz zur Event-auslösenden Instanz geben.

Ist die reagierende Instanz eine Collection-Instanz, werden alle Instanzen der Collection aktiviert, um nacheinander auf das eingetretene Ereignis reagieren zu können. Soll nur eine bestimmte Instanz der Collection reagieren, muß dies im Instanzenpfad spezifiziert werden. Noch komplexer gestaltet sich die Vorgehensweise, wenn auf dem Wege von der Event-auslösenden Instanz zur reagierenden Instanz ein oder mehrere Collections referenziert werden. In diesem Fall werden alle möglichen Kombinationen von Instanzenpfaden gebildet, um den

entsprechenden Instanzen die Reaktion zu ermöglichen. Praktisch estaltet sich das Verfahren jedoch wesentlich einfacher, als diese vielen Möglichkeiten auf den ersten Blick assoziieren.

5.4.3 Reactions nachgeordneter Instanzen

In einigen Fällen können nachgeordnete Instanzen nicht auf Ereignisse ihres übergeordneten Contexts reagieren, da es keine Rückbeziehung der untergeordneten Instanzen zur übergeordneten Instanz gibt und somit auch kein Property Path in dieser Richtung definiert werden kann (References oder Relationships ohne inverse Reference). Trotzdem ist auch hier manchmal die Reaktion auf Events des Übergeordneten Kontexts erforderlich.

> **Beispiel** Um zu sichern, daß der **Lohnfonds** der ABTEILUNG bei Änderung nicht mit die **Lohnsumme** der Mitarbeiter unterschreitet, muß ggf. infolge der Änderung des **Lohnfonds** das Aktualisieren der Gehälter der Mitarbeiter erfolgen. Dies kann nach einem vorgegebenen Algorithmus oder aber auch in einem Dialog erfolgen. Da für den Fall, daß **Mitarbeiter** der ABTEILUNG als References definiert wurden, kein Pfad von **Mitarbeiter** zu ABTEILUNG definiert werden kann, muß die Reaction als Reaction nachgeordneter Instanzen definiert werden.

Reactions nachgeordnete Instanzen können nur über eine Ebene definiert werden, d.h. es kann immer nur auf Events der direkt übergeordneten Instanz reagiert werden.

5.4.4 Reactions und Transaktionen

Im Zusammenhang mit Transaktionen entsteht die Frage, ob die Reaction auf ein Event Bestandteil der Transaktion ist, oder nicht, d.h. ob die mit dem Event verbundene Action sofort oder erst nach dem Beenden der Transaktion ausgeführt wird. Bei geschachtelten Transaktionen ergibt sich wiederum die Frage, ob die Action erst nach Beenden aller Transaktionen erfolgen soll. Wann eine Reaction erfolgen soll, ist keine Eigenschaft des Events oder der Action, sondern immer eine Eigenschaften der Reaction.

- **unmittelbare Reaktion (immediately)**

  Die unmittelbare Reaktion auf Events hat zur Folge, daß alle durch die Actions ausgelösten Zustandsveränderungen an Datenbankinstanzen Bestandteil der Transaktion sind. Wird die Transaktion zurückgenommen (**abort**), werden auch die Folgeänderungen zurückgenommen.

- **Transaktionsende (EOT)**

  Wenn die Reaktion auf Events beim Abschluß der aktuellen Transaktion erfolgt sollen, werden die Actions nur ausgeführt, wenn die Transaktion normal beendet wird (**commit**). Zu diesem Zweck werden die Reactions mit dem Event-auslösenden Objekt in eine Warteschlange gestellt, die dann nach der FIFO-Strategie abgearbeitet wird. Dabei kann allerdings nicht immer gesichert werden, daß der Zustand der Event-auslösenden Instanz der gleiche ist, wie zum Zeitpunkt des Auslösens des Events, da im Rahmen der Transaktion weitere Änderungen an der Instanz erfolgt sein können. Die Abarbeitung der Reactions erfolgt im Rahmen der zu beendenden Transaktion, so daß im Falle einer abnormalen Beendigung der Transaktion auch die Folgeänderungen zurückgenommen werden.

- **Beenden der TOP-Transaktion (EOTT)**

  Die Reactions werden erst ausgeführt, wenn die oberste Transaktion beendet wird. Dabei wird genauso verfahren, wie bei der Verarbeitung der Reactions am Ende der aktuellen Transaktion.

Werden Reactions am Ende einer Transaktion abgearbeitet, entspricht der **before**-Zustand der Event-auslösenden Instanz dem Zustand der Instanz vor Beginn der Transaktion. Der **after**-Zustand ist der Zustand am Ende der Transaktion. Da dies nicht dem Zustandsübergang beim Eintreten des Events entsprechen muß, kann das Ergebnis einer Reaction durchaus vom Zeitpunkt ihrer Ausführung abhängen. Auch die unterschiedliche Reaktion aktiver und inaktiver Instanzen spielt in diesem Zusammenhang eine Rolle, da eine Instanz aktiv gewesen sein kann, als ein Event eintrat, am Ende der Transaktion jedoch nicht mehr aktiv sein muß.

Kapitel 6 Technische Konzepte

Die Technischen Konzepte beschreiben verschieden Aspekte der physikalischen Speicherung der Daten. Diese werden hier unter folgenden Gesichtspunkten beschrieben:

- **Dateikonzept**

  Das Dateikonzept beschreibt die Dateistruktur von ODABA2-Datenbanken und die Zuordnung der Instanzen zu den Dateien.

- **Datenintegrität**

  Hier werden verschiedene Aspekte (Transaktionen, Locking-Strategien usw.) der technischen Datenintegrität in Multiuser-Umgebungen dargestellt.

- **Zugriffseffizienz**

  Prinzipelle Konzepte zur Verbesserung des Zugriffsverhaltens werden in diesem Abschnitt beschrieben.

Die Technischen Konzepte sollen somit einen Überblick über die Möglichkeiten auf der Technischen Ebene vermitteln. Sie sind jedoch keine Darstellung der technischen Realisierung des ODABA2-Datenbanksystems.

6.1 Dateikonzept

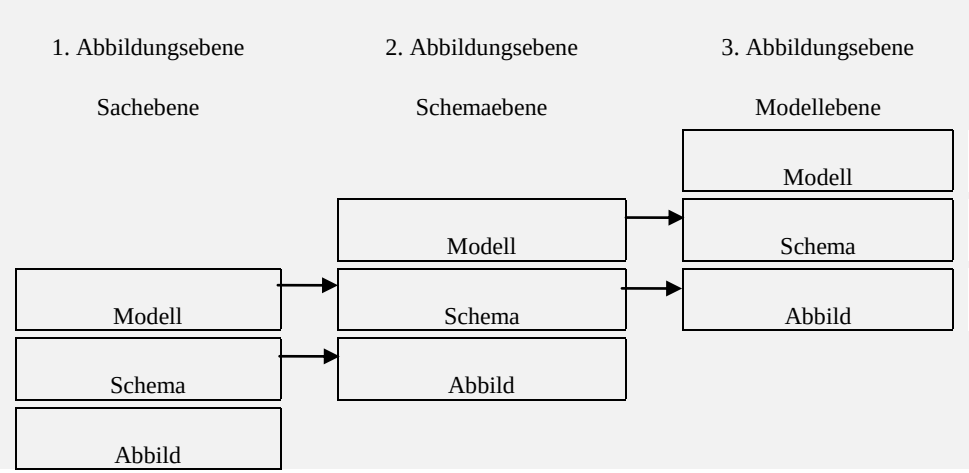
ODABA2-Datenbanken basieren auf Dateien des jeweiligen Betriebssystems. Dadurch wird eine hohe Portabilität von ODABA2-Anwendungen erreicht. Das Dateikonzept für ODABA2-Datenbanken reicht von der Kompaktdatenbank, die Daten aller Abbildungsebenen in einer physikalischen Datei speichert über eine Trennung der Datenbanken nach Abbildungsebenen bis hin zu komplexen Strukturen für verteilte Datenbanken.

Jeder Datenbank liegt ein Dictionary zugrunde, daß die Structures der Instanzen dieser Datenbank definiert. Auch hier ist es möglich, Datenbank und Dictionary in einer oder in verschiedenen Dateien zu speichern.

Die Zuordnung der Instanzen zu den verschiedenen Dateien einer verteilten Datenbank erfolgt über Instanzendeskriptoren, die die genaue Adresse einer Instanz in der Datei-Hierarchie enthalten.

6.1.1 Datenbankebenen

Das ODABA2-Datenbankmodell bildet die verschiedenen Abbildungsebenen in Datenbanken der entsprechenden Abbildungsebene ab. Aufgrund der Rekursivität des ODABA2-Modells sind dabei die Datenbanken aller Ebenen formal einfache ODABA2-Datenbanken, deren Instanzen nach den gleichen Verfahren bearbeitet werden können.



- **Modellebene (SystemDataBase)**

  Die Daten auf der Modellebene werden in einer System-Datenbank (ODE.SYS) bereitgestellt, die die Spezifikation des Datenbankmodells enthält. Eine erweiterte Modelldatenbank (ODE.RES) ist die Ressourcendatenbank der integrierten Entwicklungsumgebung, die ein erweitertes ODABA2-Modell unter Einbeziehung von Implementierungsobjekten definiert. Ebenso können andere Ableitungen des ODABA2-Modells erzeugt werden, um z.B. andere ntwicklungsmodelle zu implementieren.

- **Schemaebene (RessourceDataBase)**

  Auf der Schemaebene werden die Strukturen und kausalen Zusammenhänge einer Anwendung definiert. liegt der Schemaebene eine erweiterte Modell-Datenbank wie z.B. die erweiterte ODABA2-Modelldatenbank ODE.RES zugrunde, können auf der Schemaebene zusätzliche Modellobjekte wie z.B. Masken, Dokumente und Implementierungsklassen spezifiziert werden. Die ODE.RES selbst ist eine Ressourcendatenbank, ein Repository für die ODABA2-Entwicklungsumgebung. Die SAMPLE.RES ist eine Repository für das mitgelieferte Beispiel.

- **Sachebene (DataDataBase)**

  Auf der Sachebene werden die Structure-Instanzen und ihre Beziehungen zueinander gespeichert. Die SAMPLE.DAT ist eine Datenbank der Sachebene für das mitgelieferte Beispiel.

Die verschiedenen inhaltlichen Ebenen müssen nicht unbedingt getrennt werden. Wenn die gespeicherten Structures und Extents namentlich über alle Abbildungsebenen eindeutig sind, können Daten aller drei Ebenen in einer einzigen physikalischen Datenbank abgelegt werden. So sind z.B. in der ODE.RES sowohl die erweiterte ODABA2-Modelldefiniton (System-Datenbank) als auch Resourcen der ODABA2-Entwicklungsumgebung (Repository) gespeichert.

Jede dieser Datenbanken wird im Sinne des OODBS als unabhängige objektorientierte Datenbank aufgefaßt und kann mit den **gleichen Mitteln** bearbeitet werden, d.h. auf die Instanzen der SystemDataBase kann mit den gleichen Mitteln zugegriffen werden, wie beim Zugriff auf Instanzen der

DataDataBase. Dabei ist für den Zugriff auf eine Datenbank ein Dictionary erforderlich, das die Definition der gespeicherten Strukturen enthält.

6.1.2 Dictionary

Das Dictionary beschreibt das Objektmodell der Datenbank. In diesem Sinne ist das Dictionary i.allg. eine Teilmenge der Ressourcendatenbank oder des Repositorys, die auf einer erweiterten Modellspezifikation beruhen. Die Eigenschaft, Dictionary zu sein, ist keine Eigenschaft der Datenbank an sich, sondern drückt ihre besondere Stellung im Rahmen einer Anwendung aus. Jede Datenbank, die Structure- oder Extent-Definitionen enthält, kann im Prinzip als Dictionary eingestzt werden.

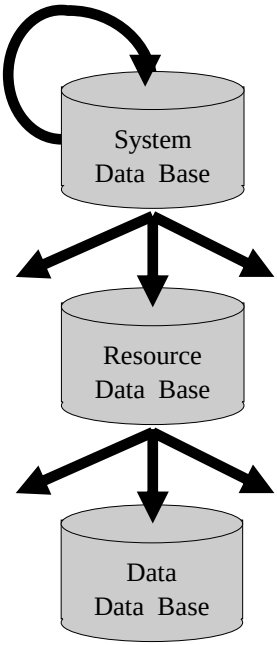
So können z.B. die ODE.SYS und die ODE.RES als Dictionaries für beliebige Ressourcendatenbanken verwendet werden, da sie die Structure- und Extent-Definitionen der Modellebene enthalten. Die SAMPLE.RES kann als Dictionary für die Beispieldatenbank SAMPLE.DAT verwendet werden. Um z.B. Ressourcen in der SAMPLE.RES zu bearbeiten, werden die Datenbanken ODE.RES und SAMPLE.RES wie folgt auf den drei Ebenen als Datenbank oder Dictionary verwendet:

Ebene	Dictionary	Datenbank
Modellebene	ODE.SYS	
Schemaebene	ODE.RES	ODE.RES
Datenebene		SAMPLE.RES

In der Beispielapplikation hingegen verändert sich die Stellung der Datenbanken, da die SAMPLE.RES hier nicht mehr Gegenstand der Bearbeitung ist, sondern als Ressourcendatenbank der Applikation fungiert:

Ebene	Dictionary	Datenbank
Modellebene	ODE.SYS	
Schemaebene	SAMPLE.RES	SAMPLE.RES
Datenebene		SAMPLE.DAT

Im allgemeinen stellt also die übergeordnete Abbildungsebene im jeweiligen Zusammenhang das Dictionary für die Bearbeitung der untergeordneten Abbildungsebene.



- **SystemDataBase**

  Die SystemDataBase stellt die Modellebene dar. Aufgrund der Rekursivität des ODABA2-Modells kann sie sowohl ihr eigenes Dictionary als auch Dictionary für ResourceDataBases sein.

- **ResourceDataBase**

  Die Ressourcendatenbank enthält alle Daten der Schemaebene. Damit kann sie als Dictionary für die Datenebene verwendet werden.

- **DataDataBase**

  In der Datendatenbank werden die eigentlichen Applikationsdaten gespeichert. Sie wird i.allg. nie als Dictionary verwendet.

6.1.3 Dateihierarchien

ODABA2-Datenbanken können auf mehrere Dateien (maximal 16 711 680) des Betriebssystems bis zu jeweils 2GB Speicher verteilt werden. Das entspricht ungefähr 32 Millionen Gigabytes. Durch die Verwendung von Dateien des jeweiligen Betriebssystems ist ODABA2 weitgehend portable zwischen verschiedenen Plattformen.

- **RootBase**

  Eine RootBase kann aus maximal 255 MainBases bestehen. Die Verteilung der Instanzen auf die MainBases erfolgt nach inhaltlichen Gesichtspunkten auf der Basis von Extents. Die MainBase 0 ist dabei identisch mit der RootBase.

- **MainBase**

  Jede MainBase kann aus maximal 256 SubBases bestehen. Die Verteilung der Instanzen auf die SubBases erfolgt nach strukturellen Gesichtspunkten. Die SubBase 0 ist identisch mit der MainBase.

- **SubBase**

  Jede SubBase kann aus maximal 256 DataAreas bestehen. Die Verteilung auf die DataAreas erfolgt nach Speicherbedarf. Die DataArea 0 ist dabei identisch mit der Datei der SubBase.

- **DataAreas**

  DataAreas können in ihrer Größe begrenzt oder dynamisch angelegt werden. Ist eine DataArea voll, werden die Instanzen in die nächste DataArea geschrieben.

Im einfachsten Fall besteht eine Datenbank aus einer RootBase-Datei, die gleichzeitig MainBase 0, SubBase 0 und DataArea 0 ist.

6.1.4 Instanzen-Identity

Da ODABA2-Datenbanken keine einfachen linearen Adreßbereiche sind, sondern über mehrere Dateien des Betriebsystems verteilt sein können, sind zur Adressierung der Instanzen mehr als vier Bytes erforderlich. Außerdem kann sich die physische Adresse einer Instanz infolge von Reorganisationsmaßnahmen oder Größenveränderung ändern, was eine Pflege aller Adreßverweise zur Folge hätte. Aus diesen Gründen erfolgt in ODABA2 eine zweistufige Identifikation der Instanzen. Die Instanzen-Identity wird durch eine interne fortlaufende Nummer dargestellt. Durch die Verwendung einer vier Bytes langen Nummer können mehr als 4 Milliarden Instanzen in einer ODABA2-Datenbank gespeichert werden. Würde eine Instanzen-Factory pro Sekunde eine Instanz erzeugen, würde es mehr als 135 Jahre dauern, um eine ODABA2-Datenbank zu füllen.

Über diese Instanzen-Identity wird ein Instanzendeskriptor adressiert, der die genaue Adresse der Instanz enthält, d.h. die Datei und die Position in der Datei. Dadurch erhöht sich zwar der Zugriffsaufwand, da vor dem Zugriff auf eine Instanz erst der Instanzendeskriptor gelesen werden muß. Andrerseits wird auf diese Weise der Speicherbedarf für die Adressierung minimiert und die Verschiebbarkeit der Instanzen gesichert. Der Instanzendescriptor wird in ODABA2 für alle freien Instanzen gebildet und ist 14 Bytes lang.

Während das erste Byte der Instanzen Identity die MainBase einer Instanz bestimmt, werden SubBase und DataArea im Instanzendescriptor abgelegt. Die Instanzendescriptoren selbst werden in der MainBase 0 gespeichert. Dadurch ist die maximale Anzahl der Instanzen in einer MainBase auf etwa 128 Millionen begrenzt.

6.1.5 Relations als externe Extents

ODABA2 läßt in begrenztem Umang die Einbindung externer Extents zu. Externe Extents sind Relationen oder Austauschformate, die als ODBC-Relationen oder Dateien bereitgestellt werden können. Externe Extents können zum einen genau wie ODABA-Extents über PI-Handle bearbeitet werden.

Beispiel Wenn Artikel-Instanzen aus einer extern gepflegten Artikel-Relation bearbeitet werden sollen, kann auch für diese Artikel-Relation ein PI-Handle erzeugt werden, wenn der Extent vorher auf der Schemaebene als externer Extent definiert wurde.

```
structure Artikel {
   attribute (  ArtNr      { type CHAR; size 12 }
                ArtName    { type CHAR; size 20 }
                ArtPreis   { type INT; size 10, precision
2 }
                ...
   )
   key          ik_Artikel  component ArtNr;
   ident_key    ik_Artikel;
}
ODBC_Extent Artikel {
   type Artikel;
   data_base Stammdat;
```

```
    relation ArtikelStamm;
}
```

Ein PI-Handle für die Artikel-Relation kann dann wie folgt erzeugt werden:

```
PI(Artikel)   art_pi(&db_handle,„Artikel“,PI_Read);
```

Über dieses PI-Handle kann in gewohnter Weise der Zugriff auf die einzelnen Artikel-Instanzen erfolgen.

Unter der Bedingung, daß ein Ident Key für die Structure definiert wurde und ein Zugriff über diesen auf die Relation möglich ist, können externe Extents auch als BaseExtents für logisch definierte Relationships oder BaseStructures verwendet werden. Auf diese Weise können sie nicht nur als Mengen in Form von Extents verwendet, sondern es ist auch möglich, Beziehungen zu Instanzen in externen Extents zu definieren.

6.2 Versionsbildung

Die Bildung von Versionen zu Instanzen kann auf der Schemaebene und auf der Datenebene erfolgen. Versionsbildung bedeutet dabei, daß der alte Zustand einer Instanz erhalten bleibt, während der neue gespeichert wird. Das ist zum einen sinnvoll, wenn die Geschichte der Änderungen einer Instanz erhalten bleiben soll, aber auch, wenn bei Update-Konflikten das Überschreiben anderer Änderungen verhindert weden soll.

Die Versionsbildung erfolgt zweistufig auf der Ebene der Real Objects und auf der Instanzenebene. Die Versionsnummer des Real Objects ist Bestandteil des Instanzendescriptors. Instanzenversionen einer Instanz werden über einen Index gespeichert, der für eine Instanzen-Identity gebildet wird, sobald diese erstmalig versioniert wird. Damit bleiben die aktuellen Instanzen weitgehend von dem Overhead verschont, der aus der Versionsverwaltung resultiert.

Bei der Versionierung einer Instanz bleibt die Instanzen-Identity für die aktuelle Instanz erhalten, während für die versionierte Instanz eine neue Instanzen-Identity gebildet wird. Instanzen-Identities für versionierte Instanzen werden nur gebildet, um auf die Instanzen verschieder Versionen zuzugreifen. Sie können nicht als

Instanzen-Referenzen in anderen Instanzen verwendet werden, d.h. der Bezug auf andere Instanzen ist immer nur über die interne Nummer der aktuellen Instanz (Instanzen Identity) möglich.

Versiondbildung für Instanzen der Datenebene

Die Versionsbildung der Daten auf der Instanzenebene erfolgt beim Speichern der Instanz. Zu diesem Zeitpunkt kann entschieden werden, ob die aktuelle Version einer Instanz überschrieben werden soll, oder ob eine neue Version der Instanz gebildet werden soll. Werden Versionen für eine Instanz gebildet, wird ein Version-Index für die Instanz angelegt. Da sich die Instanzen-Identity auch bei der Versionierung nicht ändert, wird der Version-Index jeweils für eine Instanzen-Identity erzeugt.

In dem Version-Index sind dann Verweise auf die historischen Identities, als die Instanzendeskriptoren enthalten, die die älteren, die versionierten Instanzen enthalten. In einem Version-Index kann also jede versionierte Instanz über ihre Versionsnummer erreicht werden. Dabei schließt die Versionsnummer ggf. die Object-Version mit ein.

Versionsbildung für Real Objects

Bei der Versionierung von Real Objects erfolgt automatisch eine Versionierung aller Instanzen des Real Objects, sobald sie geändert werden. Object-Versionen haben gegenüber Instanzenversionen immer den Vorrang, d.h. Instanzenversionen werden immer innerhalb einer Object-Version gebildet. Abgesehen von der Tatsache, daß die Versionsnummer dadurch gebildet wird, daß ihr die Object-Version vorangestellt wird, erfolgt die Versionierung nach den gleichen Prinzipien, wie bei der Bildung von Instanzenversionen. Die Objectversion ist Bestandteil der Versionsnummer. Innerhalb einer Object-Version beginnt die Zählung der Instanzenversionen immer mit 1. Die vollständige Versionnummer ergibt sich dann aus (Object-Version, Instanzenversion).

Da die aktuelle Object-Version außerdem im Instanzen-Descriptor vermerkt wird, ist es möglich, bei der Bildung von Object-Verionen automatisch neue

Instanzenversionen zu erzeugen, sobald Instanzen älterer Versionen geändert werden. Dadurch wird es möglich, verschiedene Versionsstände z.B. zu einem Projekt mit minimalem Speicheraufwand zu verwalten.

Versionsbildung auf der Schemaebene

Die Versionsbildung auf der Schemaebene entspricht der Versionsbildung für Entwicklungsobjekte (Projekt oder Paket). Die Versionsbildung auf der Schemaebene erfolgt nach den gleichen Prinzipien, wie auf der Datenebene. Allerdings unterscheidet sie sich in den Auswirkungen, die dies für die nachgeordnete Datenebene hat.

Da Versionsbildung auf der Schemaebene i.allg. mit einer Änderung der Sichten verbunden ist, werden die Object-Versionen der Schemaebene ebenfalls im Instanzen-Descriptor gespeichert. Dadurch ist es möglich, eine dynamische Schemaanpassung der Instanzen vorzunehmen, indem Instanzen, die eine ältere Schemaversion aufweisen, entsprechend der Schemaentwicklung über einen oder mehrere Schritte konvertiert und der neuen Structure angepaßt werden. Beim Speichern werden sie dann mit der jeweils aktuellen Schemaversion geschrieben, so daß beim nächsten Lesen keine Konvertierung mehr erforderlich ist.

Die Bildung von Object-Versionen auf der Schemaebene und die damit verbundene Konvertierung der Instanzen der Datenebene führt nicht zwangsläufig zur Bildung von Instanzenversionen auf der Datenebene. Nur, wenn mit einer neuen Schemaversion gleichzeitig neue Object-Versionen auf der Datenebene gebildet werden, wird eine entsprechende Instanzenversinierung durchgeführt. Normalerweise ist das jedoch nicht erforderlich.

6.3 Datenintegrität

ODABA2 ist in hohem Maße multiuser-fähig. Derzeit wird eine File-Server- und eine Single-User-Version angeboten. Eine Client-Server-Version auf der Instanzenebene ist in Vorbereitung. Um die Laufsicherheit der Anwendungen zu erhöhen und die Konsistenz der Daten zu gewährleisten werden auch auf der Anwendungsebene verschiedene Funktionen bereitgestellt.

- **Sperren von Instanzen**

  Das Sperren erfolgt auf der Instanzenebene. Es ist sowohl optimistisches Sperren als auch pessimistisches Sperren sowie transientes und persistentes Sperren von Instanzen möglich.

- **Transaktionen**

  Neben Systemtransaktionen werden kurze und lange Nutzertransaktionen unterstützt, die auch geschachtelt werden können.

- **Recovery-Dateien**

  Es können Recovery-Dateien geschrieben werden, die das Wiederherstellen der Datenbank im Fehlerfall ermöglichen und gleichzeitig als Änderungsprotokoll ausgewertet werden können.

6.3.1 Sperrung von Instanzen

Um parallele Änderungen zu verhindern, gibt es in ODABA2 die Möglichkeit, einzelne Structure-Instanzen oder ganze Collections für den schreibenden Zugriff durch andere Nutzer zu sperren. Diese Sperrungen werden in den meisten Fällen automatisch von den Datenbankfunktionen vorgenommen, jedoch kann es in verschiedenen Fällen auch sinnvoll sein, Structure-Instanzen explizit zu sperren.

Das Sperren von Instanzen ist nur für freie Structure-Instanzen und Collections möglich. Eingebettete Structure-Instanzen können nicht gesperrt werden. Instanten können im Rahmen eines Prozesses oder auch prozeßüberdauernd (persistent) gesperrt werden. Außerdem können Instanzen für schreibende Prozesse wie auch für lesende Prozesse gesperrt werden.

6.3.1.1 Dauerhaftes Sperren

In einigen Fällen ist es sinnvoll, einen bestimmten Strukturzustand als unveränderlich zu markieren (z.B. darf im Finanzwesen eine gebuchte Rechnung nicht mehr verändert werden).

Deshalb können in Extents oder Referenzmengen gespeicherte Strukturinstanzen dauerhaft für Änderungen gesperrt werden. Diese Sperrung überdauert den Prozeß (sie ist in diesem Sinne persistent), in dem sie ausgelöst wurde und kann später in anderen Prozessen zurückgenommen werden.

Das persistente Sperren von Instan bezieht sich nur auf ändernde Zugriffe, d.h. eine persistent geperrte Instanz ist für weitere Änderungen gesperrt (write protected). Diese Sperrung erfolgt durch eine entsprechende Markierung im Instanzendescriptor.

6.3.1.2 Zeitweiliges Sperren

Zeitweiliges Sperren (Locking) von Structure-Instanzen ist erforderlich, wenn verhindert werden soll, daß bei einer Änderung einer Instanz ein anderer Anwender Änderungen an der gleichen Instanz vornimmt oder wenn eine Instanz prinzipiell dem Zugriff anderer Applikationen zeitweilig entzogen werden soll. Gesperrt werden können Structure- und Collection-Instanzen. Die Sperrung kann implizit beim Bereitstellen der Instanz als auch explizit durch Funktionen erfolgen.

Explizites Locking

Explizites Sperren von Structure- oder Collection-Instanzen ist durch entsprechende Funktionen (Lock, LockSet) möglich. Beim expliziten Sperren wird der Instanzendescriptor gesperrt, so daß ein Zugriff auf diese Instanz aus anderen Applikationen nicht mehr möglich ist. Das schließt auch lesende Zugriffe auf die Instanz ein. Explizites Locking ist aufgrund der starken Einschränkungen nur für kurzfristige Sperrungen geeignet.

Erfolgt ein weiterer Zugriff auf eine explizit gesperrte Instanz, versucht das Zugriffssystem 10 Sekunden lang, auf die Instanz zuzugreifen. Waren alle Versuche in dieser Zeit erfolglos, wird der Zugriff mit Fehler beendet. Das Zeitintervall, in dem auf geperrte Instanzen zugegriffen werden kann, kann durch die Timeout-Variable in der INI-Datei verändert werden.

Beim expliziten Sperren ist die Applikation für das Aufheben der Sperrung zuständig. Wird dies unterlassen, wird die Sperrung erst beim Beenden der Applikation bzw. beim Schließen der Datenbank aufgehoben.

Internes Locking

Internes Locking bezieht sich ausschließlich auf das Sperren gegen schreibende Zugriffe. Internes Locking wird verwendet, um verschiedene Update-Strategien zu realisieren und Instanzen zu schützen, die in Transaktionen zwischengespeichert sind.

Internes Sperren erfolgt dadurch, daß eine virtuelle Adresse in der MainBase gesperrt wird (Lock), die der negativen Instanzen-Identity entspricht. Die Verfügbarkeit dieser Adresse wird nur für schreibende Prozesse abgefragt, so daß das Lesen der Instanz jederzeit problemlos möglich ist. Beim internen Sperren von Instanzen, die im Cluster gespeichert sind, wird nicht nur die Instanz, sondern der gesamte Cluster gesperrt. Damit ist also auf alle Instanzen des Clusters nur lesender Zugriff möglich.

- **Transaktion-Locking**

  Instanzen, die aufgrund einer Änderung in einer Transaktion zwischengespeichert werden, werden solange für andere Änderungen gesperrt, bis die Transaktion abgeschlossen ist. Soll eine geänderte Instanz in der Transaktion zwischengespeichert werden, die bereits in einem anderen Prozeß gesperrt ist, wartet das Zugriffssystem 10 Sekunden (oder die in der Timeout-Variable angegebene Zeit), um den Vorgang abzuschließen. Gelingt das nicht, wird ein Fehler erzeugt und die Transaktion ggf. abgebrochen.

  Praktisch bedeutet das, daß lange Transaktionen zu erheblichen Problemen führen können, wenn sie in Multi-User-Anwendungen eingesetzt werden und oft gemeinsam benutzte Ressourcen einschließen.

- **Pessimistisches Locking**

  Beim Pessimistischen Locking erfolgt die Sperrung automatisch, wenn eine Structure-Instanz in einem PI-Handle bereitgestellt wird, das zum exclusiven Schreiben geöffnet wurde. Nach abgeschlossener Änderung wird die Instanz gesichert und wieder freigegeben. Da beim Bereitstellen der Instanz noch gar nicht klar ist, ob überhaupt eine Änderung erfolgen wird, wird diese Strategie auch als pessimistisches Sperren bezeichnet. Es ist jedoch gegenüber der optimistischen Sperrung die sicherere Variante.

  Da während der Zeit der exclusiven Beanspruchung keiner anderen Anwendung die exclusiv schreibende Benutzung der Instanz möglich ist, ist dieses Verfahren ein sicherer Schutz gegen parallele Änderungen in verschiedenen Anwendungen. Allerdings ist der Schutz nur gewährleistet, wenn exclusive schreibende Anforderungen miteinander konkurieren, d.h., wenn alle Anwendungen nur lesen oder exclusiv schreibend auf die Instanz zugreifen. Er ist wirkungslos, wenn eine exklusiv schreibende Anforderung mit einer parallenen Update-Anforderung konkurriert.

- **Optimistisches Locking**

  Die optimistische Locking-Strategie besteht darin, daß die Structure-Instanz nicht gesperrt wird, sondern nach Abschluß der Änderung geprüft wird, ob die Instanz zwischenzeitlich von einem anderen Anwender modifiziert wurde. In diesem Fall kann die Änderung verworfen oder zwangsweise ausgeführt werden, wobei im letzteren Fall die Änderungen des anderen Nutzers überschrieben werden. Die Prüfung auf Änderung durch andere Applikationen erfolgt, wenn die Instanz als geändert in die Transaktion gesichert wird. Solange sie in der Transaktion zwischengespeichert ist, ist die Instanz gegen andere Änderungen gesperrt.

  Optimistisches Locking ist nur für Instanzen in geteilt benutzbaren References (Extents oder so markierte References in Structure-Instanzen) möglich.

6.3.2 Transaktionen

Transaktionen werden gebildet, um mehrere Instanzenänderungen eines Prozesses zusammenzufassen. Da es oft kompliziert ist, bereits vollzogene Änderungen in

einem Prozeß zurückzunehmen, arbeiten Transaktionen nach dem Prinzip, daß alle Änderungen erst einmal temporär vollzogen werden. Sie werden also nicht wirklich in der Datenbank gespeichert, sondern in einem Transaktionspuffer abgelegt. Aus der Sicht der Anwendung ist kein Unterschied sichtbar, da jeder Zugriff zuerst im Transaktionspuffer sucht, bevor er in der Datenbank liest. Für andere Anwendungen sind die Änderungen jedoch nicht sichtbar, solange die Transaktion noch nicht abgeschlossen ist. Erst wenn die Transaktion beendet wird, werden die Instanzen aus dem Transaktionspuffer in die Datenbank zurückgeschrieben. Muß die Transaktion abgebrochen werden, entfällt das Aktualisieren der Datenbank, und der Zustand ist derselbe, wie am Beginn der Transaktion. Während der Dauer einer Transaktion werden in anderen Applikationen die noch nicht geänderten Instanzen der Datenbank bereitgestellt.

Dieses an sich sehr vorteilhafte Verfahren besitzt jedoch auch seine Tücken. Da Transaktionen nur im Ganzen angenommen oder verworfen werden können, werden die Instanzen, die im Rahmen einer Transaktion geändert wurden, bis zum Ende der Transaktion gesperrt (Transiente Lock). Das bedeutet, daß sich insbesondere für Collections die Sperrintervalle vergrößern. Deshalb sollte vor allem das Hinzufügen oder Löschen in Extents nicht oder nur in kurzen Transaktionen erfolgen. Das zweite Problem besteht in der Sicherung der logischen Konsistenz. Da das Prüfen der Änderungen und das Schreiben der Instanzen auseinanderfallen, können andere Prozesse in dieser Zeit Änderungen auf der Basis der alten Instanzenzustände durchführen, die nach Abschluß der Transaktion nicht mehr gültig sind.

In ODABA2 werden drei verschiedene Transaktionsmechanismen unterstützt:

- **interne Transaktionen**

  Jeder Funktionsaufruf zur Datenmanipulation (PI-Funktionen) startet eine interne Transaktion. Dadurch können auch Folgeänderungen zurückgenommen werden, die durch Reactions im Rahmen einer Änderung ausgelöst wurden. Bei erfolgreicher Ausführung der Funktion wird die Transaktion bei Beenden der Funktion gespeichert. Ansonsten werden alle im Rahmen der Funktion ausgeführten Änderungen verworfen.

Interne Transaktionen werden nur durch das Zugriffssystem gestartet und beendet. Aus der Anwendung heraus können keine internen Transaktionen gestartet werden.

- **kurze Transaktionen**

  Durch kurze Transaktionen ist es möglich, eine Menge von inhaltlich zusammengehörenden Änderungen aus Sicht der Applikation in einer Transaktion zusammenzufassen. Kurze Transaktionen sind Transaktionen, bei denen die geänderten Instanzen in einem Transaktionspuffer zwischengespeichert werden. Erst, wenn die Transaktion erfolgreich beendet wird, erfolgt das Schreiben der in der Transaktion zwischengespeicherten Instanzen in die Datenbank (bzw. in die übergeordnete Transaktion). Wird ein Fehler festgestellt, kann die Transaktion vollständig verworfen werden.

- **Lange Transaktionen**

  Lange Transaktionen fassen ähnlich wie kurze Transaktionen eine Menge inhaltlich zusammengehöriger Änderungen zusammen. Im Gegensatz zu kurzen Transaktionen werden diese jedoch nicht in einem internen Puffer, sondern in einer Transaktion-Base zwischengespeichert, die als temporäre RootBase vom Zugriffssystem angelegt wird. Dadurch ist die Beschränkung des Transaktionsumfangs durch Hauptspeichergrenzen aufgehoben. Allerdings sind lange Transaktionen auch deutlich langsamer als kurze.

geschachtelte Transaktionen

Unabhängig von ihrem Art können Transaktionen geschachtelt werden. Dabei werden Änderungen nach Beenden einer Transaktion nicht sofort in die Datenbank, sondern erst in die übergeordnete Transaktion übernommen. Erst das Beenden der Top-Transaktion führt dazu, daß die Änderungen in die Datenbank übernommen werden. Auf diese Weise können Teilaktivitäten im Rahmen umfangreicherer Transaktionen geschlossen zurückgenommen werden.

Parallele Transaktionen

Haüfig ist es sinnvoll, parallele Änderungen in einer Applikation inhaltlich zusamenzufassen. Das kann über geschachtelte Transaktionen nicht gelöst werden,

da für geschachtelte Transaktionen die übergeordneten Transaktonsintervalle die untergeordneten zeitlich vollständig einschließen. Gibt es aber Überlappungen von Vorgängen oder werden Aktivitäten unterschiedlicher Vorgänge versetzt ausgeführt, können diese nur in parallelen Transaktionen zusammengefaßt werden. Parallele Transaktionen können auch im Rahmen einer geschachtelten Transaktion gestartet werden, um innerhalb dieser parallel laufende Prozesse zusammenzufassen.

6.3.3 Log- und Recovery-Datei

Die an einer Datenbank vorgenommenen Änderungen können in einer Logdatei gespeichert werden. Damit sind zwei Vorteile verbunden. Der erste besteht darin, daß in jedem Logdateisatz vermerkt wird, von welchen Nutzer zu welcher Zeit eine Änderung vorgenommen wurde. Dadurch wird eine Kontrolle der Nutzeraktivitäten bei Störfällen möglich.

Zum zweiten kann diese Datei verwendet werden, um im Havariefall die Datenbank von letzten Sicherungszeitpunkt ausgehend zu aktualisieren.

Transaktionen werden nur in die Logdatei übernommen, wenn die Top-Transaktion erfolgreich abgeschlossen wurde. Dabei erfolgt zuerst das Schreiben in die LOG-Datei, bevor dir Transaktion in der Datenbank gespeichert wird. Beim Rückspeichern ber Instanzen aus der LOG-Datei werden nur die Instanzen abgeschlossener Transaktionen übernommen.

LOG-Dateien können lokal (Anwendungsbezogen) oder Global (datenbankbezogen) erzeugt werden. Bei der Verwendung lokaler LOG-Dateien müssen beim Recovery alle LOG-Dateien bereitgestellt werden. Die Synchronisation der Instanzen in den verschiedenen LOG-Dateien erfolgt dabei anhand der Zeit (Datum/Uhrzeit) und anhand der Modifikationsnummer im Instanzendescriptor.

6.4 Zugriffseffizienz

Um die Zugriffseffizienz auf die Instanzen zu verbessern, werden sowohl Puffertechniken als auch die Bildung von Clusters zur geschlossenen Speicherung mehrer Instanzen implementiert.

6.4.1 Pufferung

Um Effizienzverlußte, die u.a. durch den mehrstufigen Zugriff über die Instanzen-Identity auf die Instanz entstehen, auszugleichen, wurden in ODABA2 verschiedene Techniken zur Effizienzsteigerung implementiert.

6.4.1.1 Instanzenpuffer

ODABA2 verfügt über einen internen Instanzenpuffer, der nicht nur einen effizienten Zugriff auf die Instanzen sichert, sondern auch die Eindeutigkeit der Instanzen gewährleistet. In einer ODABA2-Applikation werden alle Instanzen in Structure-bezogenen Puffern intern zwischengespeichert. Über diese Puffer wird auch die Eindeutigkeit der Instanzen im Rahmen einer Applikation gesichert. Allerdings ist dieses interne Instanzenformat nicht identisch mit dem Format, das in einer Programmiersprache gefordert wird, zumal die verschiedenen Programmiersprachen die Instanzen unterschiedlich darstellen. Aus diesem Grunde werden die gepufferten Instanzen in ein entsprechendes Programmiersprachenbezogenes Format (Standardmäßig C++) konvertiert. Dabei können beliebig viele Duplikate einer Instanz erzeugt werden.

Über die Structure-Buffer werden jedoch Änderungen registriert, so daß Änderungen von verschiedenen Stellen einer Applikation festgestellt und behandelt werden können. Zeitweilig können jedoch Inkonsistenzen zwischen der konvertierten und der gepufferten Instanz auftreten.

Structure-Puffer werden auf jeder Transaktionsebene (ausgenommen interne Transaktionen) angelegt. Dadurch kann es bei parallelen Transaktionen geschehen, daß in parallelen Transaktionen verschiedene Instanzenzustände in einer Applikation bearbeitet werden.

6.4.1.2 Transaktionspuffer

Bei Verarbeitung großer Datenmengen kann die Effizienz durch die Verwendung von dynamischen Transaktionspuffern erheblich gesteigert werden. Dynamische Transaktionspuffer sind interne Transaktionen, die jedoch nicht nach inhaltlichen Kriterien abgeschlossen werden, sondern geschrieben werden, wenn ein bestimmter Füllungsgrad erreicht ist.

Transaktionspuffer werden vor allem zur Speicherung von größeren Datenmengen verwendet. Damit wird nicht nur die Zugriffseffizienz erhöht, sondern auch das wiederholte Schreiben wachsender Indizes verhindert, was zu einer wesentlich verbesserten Speicherauslastung führt.

Dynamische Transaktionpuffer für 50-200 Instanzen können schon zu einer erheblichen Verbesserung des Laufzeitverhaltens führen. Da Instanzen, die im Dynamischen Transaktionpuffer zwischengespeichert werden, füe andere Applikationen gesperrt sind, sollten die Schreibintervalle, also der Zeitraum, den der Transaktionpuffer zum Füllen benötigt, 1-2 Sekunden nicht überschreiten.

6.4.2 Clusterbildung

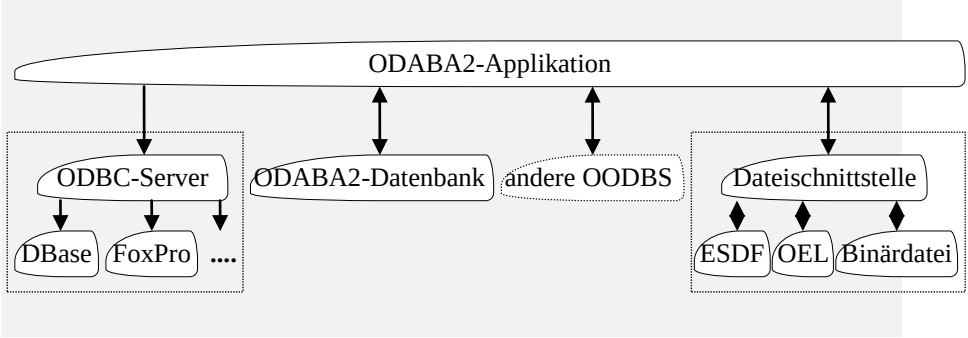
Durch die Bildung von Clustern können Instanzen nach inhaltlichen Kriterien in einem gemeinsamen Speicherbereich untergebracht werden. Indem Cluster als eine Einheit gelesen und geschrieben werden, kann die Zugriffseffizienz für Instanzen, die oft zusammenhängend verarbeitet werden, entschieden verbessert werden. Die geclusterte Speicherung ist vor allem zur Speicherung kleiner Instanzen mit enger Bindung an die übergeordnete Instanz geeignet.

Die geclusterte Speicherung einer Instanz führt ersteinmal dazu, daß alle Indizes von referenzierten nicht geteilt benutzbaren Collections gemeinsam mit der Instanz im Cluster gespeichert werden. Darüber hinaus kann für verschiedene References angegeben werden, daß auch die referenzierten Instanzen im Cluster gespeichert werden sollen. Die Größe eines Clusters ist derzeit auf 64 KB begrenzt. Ist diese Grenze erreicht, werden die Instanzen außerhalb des Clusters abgelegt.

Allerdings ist die Bildung von Clusters auch mit Nachteilen verbunden, da Instanzen in einem Cluster nicht mehr einzeln gesperrt werden können. Mit der Sperrung einer Instanz in einem Clusters wird also immer der gesamte Cluster gesperrt.

Kapitel 7 Schnittstellen

ODABA2 unterstützt verschiedene Schnittstellen zu relationalen und objektorientierten Systemen. Dazu gehören allgemeine Schnittstellen wie ODBC genauso, wie Austauschformate auf der Dateiebene (ESDF, OEL). Vorgesehen ist außerdem die direkte Unterstützung anderer objektorientierter Datenbanken wie z.B. POET oder ObjectStore. Dadurch wird zum einen der direkte Zugriff auf andere Systeme möglich. Zum anderen können Daten nach verschiedenen Gesichtspunkten in unterschiedlichen Quellen abgelegt werden.



Datenquellen für ODABA2-Anwendungen

Auf alle Datenquellen und Schnittstellen kann über PI-Handle zugegriffen werden, wobei die Funktionalität entsprechend der Schnittstelle eingeschränkt ist. Letztlich macht es jedoch innerhalb einer Anwendung keinen Unterschied, ob die Daten aus einer ODABA2-Datenbank oder über die ODBC-Schnittstelle bzw. aus einer Datei gelesen werden. Dateien und Relationen werden dabei jeweils als Extents mit einer Menge gleichartiger Objekte aufgefaßt.

Die Kommunikation zwischen ODABA2 und anderen Systemen wird auf drei Ebenen unterstützt:

- **Import/Export-Schnittstellen zu anderen Systemen**

  Eine Möglichkeit ist der Datenaustausch mit anderen Systemen auf der Basis relationaler Sichten über Import/Export-Funktionen von ODABA2. Dabei werden drei Dateiformate unterstützt, über die Daten mit objektorientierten oder relationalen Systemen ausgetauscht werden können. Außerdem können über die ODBC-Schnittstelle Daten mit allen Datenbanksystemen ausgetauscht werden, die ODBC unterstützen. ODABA2 kann dabei zum einen aktiv auf andere Datenbanken über deren ODBC-Schnittstelle zugreifen. Zum anderen stellt ODABA2 selbst eine ODBC-Schnittstelle bereit, über die andere Systeme lesend auf ODABA2-Daten zugreifen können.

- **Datenaustausch mit ER-Systemen**

  Da bei dem Datenaustausch über relationale Sichten i.allg. die Beziehungen des ER-Modells verloren gehen, stellt ODABA2 komplexe Import/Export-Möglichkeiten zum Datenaustausch mit ER-Systemen bereit. Dabei können Zusammenhänge eines ER-Modells vollständig in ODABA2 importiert oder aus ODABA2 exportiert werden.

- **Direktzugriff auf andere Datenbanken**

  Während der Datenaustausch auf einer Tabellenorientierten Sicht basiert, unterstützt der Direktzugriff auf andere Datenbanken eine Instanzen- oder Tupel-bezogene Sicht auf die Daten. Der Direktzugriff ist zur Zeit auf Binärdateien, SQL-Server-Datenbanken und Datenbanksysteme mit ODBC-Schnittstelle möglich. Die Unterstützung weiterer relationaler und objektorientierter Systeme ist geplant.

7.1 Import/Export auf der Basis relationaler Sichten

Der Relationen-basierte Import/Export von Daten wird sowohl durch verschiedene Datei-Formate als auch über ODBC-Schnittstellen unterstützt. Damit ist es zum einen möglich, Daten Off-Line zwischen verschiedenen ODABA2-Anwendungen auszutauschen. Andrerseits können Daten auch mit anderen Systemen ausgetauscht werden, wenn diese eines der folgenden Formate unterstützen.

- **Binärdateien (BINA)**

  Binärdateien stellen die Daten in Form strukturierter Sätze bereit, die dem Aufbau einer Objektdatei entsprechen. Über dieses Format können i.allg. nur relationale Strukturen ausgetauscht werden.

- **Extended SDF (ESDF)**

  Extended SDF (Self Delimiter File) sind eine Erweiterung des relationalen SDF-Formates. Über ESDF können Instanzen mit relationalen sowie mit komplexen Objektstrukturen ausgetauscht werden.

- **Object Exchange Language (OEL)**

  Die OEL ermöglicht die Darstellung komplexer Zusammenhänge in einer ASCII-Datei. Über OEL können Daten der Daten- und der Schemaebene ausgetauscht werden.

- **ODBC-Schnittstellen**

  Mit Datenbanken, die eine ODBC-Schnittstelle unterstützen, kann der Datenaustausch direkt über diese ODBC-Schnittstelle erfolgen. Dabei wird sowohl der Import als auch der Export der Daten unterstützt.

Alle externen Datenquellen für den Import/Export werden von ODABA2 als Extents mit speziellem Zugriffstyp behandelt. Damit können sie prinzipiell auch wie ganz normale ODABA2-Extents behandelt werden. Aufgrund ihrer Ausrichtung auf Datei- bzw. Tabellenstrukturen ist jedoch die direkte Einbindung dieser Extents nicht empfehlenswert. Durch einfache Copy-Funktionen ist es jedoch möglich einen „externen“ Extent in einen ODABA2-Extent zu übernehmen oder die ODABA2-Daten in den externen Extent zu übertragen.

7.1.1 Binärdateien

Binärdateien erlauben die Darstellung der Daten im Binärformat ebenso wie im ASCII-Format. Die einzige Voraussetzung ist, daß die Dateien aus Sätzen mit fester Struktur (und gleicher Länge) bestehen. Die Property-Instanzen einer Structure-Instanz sind dann durch die Position im Satz bestimmt.

In Structure-Definitionen für Binärdateien können sowohl komplexe Attributes als auch Base Structures, singuläre Relationships und References dargestellt werden. Binärdateien sind die einfachste Möglichkeit des Datenaustauschs, solange die Instanzen keine nachgeordneten Mengen (References) besitzen.

Binärdateien werden als entsprechende Extents definiert, wobei der Dateiname Bestandteil der Extent-Definition ist. Wie auf andere Extents wird auch zum Zugriff auf BINA-Extents ein PI-Handle konstruiert, über welches lesende und schreibende Zugriffe auf die Datei erfolgen.

Beispiel Um Artikeldaten eines Liefernden, die als strukturierte Datei (BINA-File) vorliegen, in eine ODABA2-Anwendung zu importieren, muß auf der Schemaebene die Structure der Sätze (EXTARTIKEL) und ein BINA-Extent (**ExtArtikel**) definiert werden, der die Dateispezifikation enthält. Dann kann ganz normal ein PI-Handle geöffnet und die Instanzen importiert werden.

```
   PI(ExtArtikel)
extart_pi(dbhandle,"ExtArtikel",PI_Read);
   PI(Artikel)    art_pi(dbhandle,"Artikel",PI_Read);
   art_pi.CopySet(extart_pi);
```

Ebenso kann das Importieren über entsprechende Funktionen der ODABA2-Datenbankdienste erfolgen.

7.1.2 Extended SDF (ESDF)

Extended SDF ist eine Erweiterung des Standardformates SDF, welche die Darstellung komplexer Zusammenhänge zuläßt und somit wesentlich mehr Möglichkeiten zur Darstellung von Objekten enthält, als das SDF-Format an sich. Das ESDF-Format ermöglicht über die Möglichkeiten des relationalen Datenaustausches den Datenaustausch komplexer Objekte zwischen Objektorientierten Systemen. Im ESDF-Format können neben komplexen Instanzen auch Mengen von Instanzen (Collections) oder Beziehungen (Relationships) ausgetauscht werden.

ESDF-Dateien sind hauptsächlich für den Datenaustausch konzipiert und unterstützen im wesentlichen eine sequentielle Verarbeitung. Sie können zwar als auch externe Extents geöffnet werden, sind aber bezüglich ihrer Zugriffsfunktionalität stark eingeschränkt.

Das setzt in einigen Fällen allerdings die Möglichkeit der Identifizierbarkeit der Instanzen voraus. Da die Instanzen-Identity sich zwar zur Identifikation einer Instanz innerhalb eines Systems, nicht aber des durch die Instanz dargestellten Sachverhaltes eignet, ist der Austausch von Daten in vielen Fall nur über inhaltliche Kriterien möglich, z.B. indem zwischen den Systemen ein identifizierender Schlüssel vereinbart wird.

Für einfache relationale Strukturen entspricht das ESDF-Format dem SDF-Format. Relationale Strukturen werden dadurch dargestellt, daß die Attribute durch einen definierten Property Delimiter (in den Beispielen wird **;** als Feldtrennzeichen verwendet) voneinender getrennt in der Reihenfolge ihrer Definition als ASCII-Zeichenfolge dargestellt werden.

Beispiel Eine ANSCHRIFT-Instanz, bestehend aus den Attributes **strasse**, **hausnummer**, **PLZ** und **ort** würde im SDF ebenso wie im ESDF-Format wie folgt dargestellt werden:

```
'Koepenicker Str';325;12555;Berlin;
```

Undefinierte Werte am Ende einer Instanz können dabei weggelassen werden. Das Ende der Instanz wird durch das Zeilenende (LF/CR) oder durch definierte Structure-Delimiter markiert. Das ESDF erlaubt darüberhinaus jedoch auch die Darstellung komplexer Properties und References. Komplexe Properties werden durch definierte Structure Delimiter (hier **{ }**) begrenzt. Innerhalb dieser Klammern können Properties einer Structure-Ebene angegeben werden. Referenzierte oder eingebettete Structure-Instanzen (dazu gehören auch Base Structure-Instanzen), werden dann jeweils in Structure Delimiter eingeschlossen.

Beispiel Um die Adresse in einer FIRMA-Instanz betehend aus den Attributes **name**, und **adresse** würde im ESDF-Format darzustellen, wäre folgende Angabe möglich:

```
'run Software-Werkstatt GmbH';{'Koepenicker Str';325;12555;
Berlin};
```

Die zweite Erweiterung des ESDF-Formates gegenüber dem SDF-Format besteht in der Angabe von Collections, die eine Menge von gleichartigen Elementen einschließen. Dadurch können sowohl Array-Elemente als auch Elemente von Reference-Collections dargestellt werden. Die Elemente der Menge werden in

Structure Delimiter (hier **{ }**) eingeschlossen. Sind die Elemente elementarer Art, können die Structure Delimiter entfallen.

Beispiel Eine PERSON-Instanz bestehend aus den Attributes **name**, und **vorname[3]** würde im ESDF-Format wie folgt aussehen:

```
Müller;{Anton} {Otto};          oder
Müller;Anton Otto;
```

Da *Anton* nur zwei Vornamen hat, kann die Angabe des dritten Namens entfallen. Generell können leere Instanzen aber durch **{}** angegeben werden.

Die Trennzeichen des ESDF sind frei wählbar, so daß Konflikte mit anderen Zeichen in den Daten weitgehend vermieden werden können. Die folgende Zusammenfassung definiert die Syntax der ESDF unter Verwendung der in den Beispielen benutzten Trennzeichen.

instance := *prop_inst*[12] [;[13] *prop_inst* ...] [;][14]

prop_inst := **{** *instance* **}**
prop_inst [*prop_inst* ...]
value

value := Folge von ASCII-Zeichen, die keines der definierten Trennzeichen oder Leerzeichen enthält. Zeichenketten müssen durch ein definiertes Zeichen geklammert werden (z.B. ‘), wenn sie Trennzeichen oder Leerzeichen enthalten oder mit einer Ziffer beginnen. Werte mit 0 Zeichen werden als nicht vorhandene Werte betrachtet.

Diese einfachen Regeln erlauben die Darstellung aller strukturellen Zusammenhänge in einer objektorientierten Datenbank. Allerdings können in einer ESDF-Datei nur Instanzen eines Extents und von diesen referenzierte Instanzen dargestellt werden. Außerdem ist die Darstellung der Daten im ESDF-Format oft mit Redundanzen verbunden, da mehrfach referenzierte Instanzen auch mehrfach im ESDF dargestellt werden. Vor allem rekursive Bezüge (z.B. durch inverse References für Relationships) führen oft zu endlosen Rekursionen und müssen deshalb bei der Definition von ESDF-Formaten vermieden werden. Formale Rekursionen, die sich aus den beidseitigen References in Relationships ergeben,

---

[12] Kursiv gedruckte Begriffe bezeichnen komplexe Ausdrücke, die im folgenden definiert werden.

[13] Fettgedruckte Zeichen beschreiben konstante Sprachelemente.

[14] Das Ende einer Instanz wird auch bei Zeilenumbruch oder abschließenden Structure Delimiter erkannt.

werden dadurch vermieden, daß eine der beiden References als sekundäre Reference markiert wird. Bei der Erzeugung von ESDF-Instanzen zum Exportieren werden sekundäre References nicht abgebildet.

7.1.3 Object Exchange Language (OEL)

Im Gegensatz zum ESDF-Format, das ein geeignetes Austauschformat darstellt, ist die OEL mehr eine Instanzen-Beschreibungssprache. Die Prinzipien der OEL sind ähnlich wie die der ESDF. So sind auch OEL-Dateien hauptsächlich für den Datenaustausch konzipiert. Der wesentliche Unterschied besteht darin, daß die Daten nicht über ihre Position im Satz, sondern über die Property-Namen den entsprechenden Properties zugeordnet werden. Dadurch eignet sich das OEL-Format besonders zum partiellen Import/Export von Daten. Aufgrund ihres Umfangs ist die OEL allerdings nicht so gut zum Import/Export großer Datenmengen geeignet.

Der Datenabschnitt einer OEL-Datei wird durch das Schlüsselwort DATA eingeleitet und kann Daten zu einem oder mehreren Extents enthalten. OEL-Dateien können neben den Daten auch die Metadaten (Structure-Definitionen) enthalten. Diese werden in einem Abschnitt am Anfang der OEL-Datei abgelegt, der durch das Schlüsselwort METADATA eingeleitet wird. Hier werden sowohl die Structure als auch die Extent-Definitionen der gespeicherten Instanzen und Extents abgelegt. Die Structure- und Extent-Definitionen folgen der ODABA2-Semantik, können aber durch einen definierten DataExchange auf der Schemaebene in jede andere Semantik konvertiert werden.

Durch die Verwendung der Property-Namen werden außerdem die strukturellen Abhängigkeiten reduziert. Dadurch, daß in der OEL die Properties nur an ihren Namen gebunden sind, ist ein Austausch von Instanzen auch möglich, wenn die Instanzen unterschiedliche Strukturen aufweisen. Die Verwendung gleichartiger Begriffe auf der Schemaebene vorausgesetzt, wird durch die OEL der Austausch von Daten somit zwischen beliebigen OODBS und Anwendungen möglich, auch wenn die Structures bezüglich ihrer Properties nicht vollständig übereinstimmen.

Da das ODM rekursiv ist, eignet sich die OEL genauso gut zur Darstellung von Schemadefinitionen, d.h. die OEL kann als Object Definition Language verwendet werden, wodurch der Austausch von Instanzen der Schemaebene (z.B. Structure-Definitionen) möglich wird. Allerdings sind Schemadefinitionen zwischen verschiedenen OODBS erst dann austauschbar, wenn hinsichtlich der Begrifflichkeit und der Struktur der Schemaebene Einigkeit besteht, also z.B. zwischen zwei ODABA2-Applikationen.

Die folgenden Beispiele illustrieren die wesentlichen Prinzipien, die der OEL zugrunde liegen. Im Gegensatz zum ESDF-Format werden Structure-Instanzen dadurch dargestellt, daß ihre Properties benannt und mit Werten verbunden werden. Die Properties werden dabei durch Property Delimiter (**;**) voneinander getrennt.

Beispiel Eine ANSCHRIFT-Instanz, bestehend aus den Attributes **strasse**, **hausnummer**, **PLZ** und **ort** würde mittels der OEL wie folgt dargestellt werden:

```
{ strasse 'Koepenicker Str'; hausnummer 325; PLZ 12555;
ort Berlin }
```

Undefinierte Werte oder Werte, die nicht verändert werden sollen, können einfach weggelassen werden. Auch die OEL erlaubt die Darstellung komplexer Properties und References. Komplexe Properties werden wieder durch Structure Delimiter (hier **{ }**) begrenzt. Innerhalb dieser Klammern können Properties einer Structure-Ebene angegeben werden. Referenzierte oder eingebettete Structure-Instanzen werden dann jeweils in Structure Delimiter eingeschlossen.

Beispiel Die Adresse in einer FIRMA-Instanz bestehend aus den Attributes **name**, und **adresse** würde im OEL-Format wie folgt dargestellt werden:

```
{  name „run Software-Werkstatt GmbH";
   adresse { strasse „Koepenicker Str"; hausnummer 325;
             PLZ 12555; ort Berlin } }
```

Instanzen von Collections werden in der OEL als Folge von Instanzen dargestellt, die jeweils durch Leerzeichen getrennt sind. Komplexe Instanzen sind dabei wieder in Structure Delimiter einzuschließen.

Beispiel Eine PERSON-Instanz bestehend aus den Attributes **name**, und **vorname[3]** würde im OEL-Format wie folgt aussehen:

```
name Müller; vorname Anton Otto;
```

Eine Menge von PERSONEN-Instanzen könnte wie folgt definiert werden:

```
Person {name Müller} {name Smith} {name Richard};
```

Beim Austausch globaler Instanzen, die i.allg. durch einen IdentKey identifiziert werden, sollte gesichert werden, daß die Instanzen wenigstens die IdentKey-Angaben enthalten. In einigen Fällen ist es darüber hinaus erforderlich, die Instanzen in einer Menge zu positionieren. Zu diesem Zweck können Collection-References mit einem Index versehen werden.

Beispiel Eine PERSON-Instanz bestehend aus den Attributes **name**, und **vorname[3]** würde im OEL-Format wie folgt aussehen:

```
name Müller; vorname[0] Anton; vorname[2] Otto;
```

In diesem Fall würde der erste und der dritte Vorname in die entsprechende Personen-Instanz übernommen werden. Der zweite Vorname bliebe unverändert.

In einer OEL-Datei können Instanzen verschiedener Art gespeichert werden. Dazu werden in der OEL-Datei genau wie in der Datenbank verschiedene Extents definiert. Der Extent-Bezug kann jedoch genauso gut in der Extent-Definition der OEL-Datei festgelegt werden, wenn die OEL-Datei nur Instanzen eines Extents enthält.

Beispiel In einer OEL-Datei können Instanzen mehrerer Extents gespeichert werden. Dabei können auch Beziehungen zwischen den Instanzen dargestellt werden.

```
Firma   { name 'run Software-Werkstatt GmbH';
          adresse{ strasse 'Koepenicker Str';
                   hausnummer 325; PLZ 12555; ort Berlin
} }
        { name 'Informatics Consulting GmbH' };

Person  { pid 000001; name Mueller;
          vorname Anton Otto;
          firma { name 'run Software-Werkstatt GmbH'}
};
```

Hier ist über **firma** in PERSON eine Relationship zu FIRMA hergestellt worden. Die inverse Reference in Firma (**mitarbeiter**) ist in diesem Fall nicht angegeben, würde aber beim Importieren dieser OEL-Datei automatisch hergestellt werden.

Die OEL beruht auf der alternierenden Hierarchie von Property- und Structure-Instanzen. Da jeder Property eine Structure und jeder Structure eine Property

übergeordnet ist, geht es jeweils darum, im Structure oder Property Kontext eine Instanz zu identifizieren. Property-Instanzen werden im Kontext einer Structure durch ihren Namen festgelegt. Jede Property-Instanz beginnt mit ihrem Property-Namen und endet i.allg. mit einem Property Delimiter (hier **;**). Eine Property kann aus einer oder mehreren Structure-Instanzen bestehen. Enthält eine Property-Instanz als Collection mehrere Instanzen, werden diese im Property Kontext durch ihre Instanzen-Identity, ihre Position in der Collection oder durch ihre IdentKey-Instanz bestimmt. Eine OEL-Anweisung hat somit folgenden Aufbau:

```
oel_file   :=   [METADATA instance;] [DATA instance;]

instance   :=   value
                {statement [statement..]}

statement  :=   property  instance;
                property[ident] instance;

ident      :=   ganzzahliger Wert (instanzenposition).

value      :=   Zeichenkette (Wert für elementaren Typ)
```

- **property – zu definierender Sachverhalt**

  Jede Instanz existiert in einem Property-Kontext, der durch den Property-Namen festgelegt ist. Dies kann sowohl eine freie Property einer Datenbank oder eines Objects (Extent oder freie Collection) als auch die Property einer Structure sein. Eine Property kann mit einer oder mehreren Property-Instanzen verbunden sein.

- **instance – Definition von Instanzen**

  Die Definition einer Instanzen basiert auf den Properties des Types, dem die Instanz entspricht. Für Basic Types und Enumerations besteht die Definition einfach in der Spezifikation eines Wertes (**value**). Für komplexe Types sind die Properties der Structure zu definieren (**statement** oder Folge von **statements**). Es müssen jedoch nicht alle Properties der Structure angegeben werden. Nicht definierte Properties werden nicht verändert oder mit Standardwerten initialisiert.

- **ident - Position der Instanz**

  Die erste Instanz einer Collection hat immer die Nummer 0. Für statische Bereiche (Arrays) muß der Wert zwischen **0** und **Bereichsdimension-1** liegen. Für dynamische Collections wird die Instanz an der angegebenen Position ersetzt oder eine neue Instanz am Ende der Collection angefügt (wenn die angegebene Position größer oder gleich der Anzahl der Instanzen in der Collection ist).

- **value - Wert der Instanz**

  Folge von ASCII-Zeichen, die keines der definierten Trennzeichen enthält. Zeichenketten müssen nur in Delimiter (z.B. ‘) eingeschlossen werden, wenn sie Leerzeichen oder definierte Delimiter enthalten bzw. Mit einer Ziffer beginnen. Ist keine Zeichen für den Wert angegeben, wird die entsprechende Property-Instanz auf den leeren Wert initialisiert.

7.1.4 ODBC-Schnittstelle

Über die ODBC-Schnittstelle ist der Zugriff auf „relationale Extents“ möglich. Voraussetzung dafür ist, daß die Relation, die als Extent verarbeitet werden soll, über die ODBC-Schnittstelle angesprochen werden kann. Jedes Datenbanksystem, das über ODBC eine relationale Sicht auf die Daten ermöglicht, kann auf diese Weise von ODABA2 verarbeitet werden. Allerdings müssen die Namen der Attributes der Relation mit denen, die für die ODABA2-Structure definiert wurden, übereinstimmen, da die Instanzen für relationale Extents nur die Attributes enthalten, die in der ODABA2-Structure definiert sind.

Statt auf eine Relation ist auch der Zugriff auf eine SQL-View möglich. Diese wird ebenfalls wieder als eine Relation aufgefaßt. Der Umfang der View-Definition richtet sich dabei nach der Leistungsfähigkeit des ODBC-Treibers der angesprochenen Datenbank.

In begrenztem Umfang ist auch das Schreiben in Relationen möglich. Dabei sind die Grenzen wieder durch die jeweiligen ODBC-Treiber gesetzt. Dabei gibt es vor

allem hinsichtlich der Möglichkeiten der Änderung von Views Einschränkungen. Einige Systeme lassen überhaupt nur lesende Zugriffe zu.

ODBC-Relationen werden in ODABA2 als Extents (ODBC-Extents) definiert. Die Extent-Definition enthält neben der Spezifikation der Relation (Server- und Relation-Name) einen Bereich, in dem SQL-Statements angegeben werden können. Werden vom Anwender keine SQL-Statements angegeben, wird ein Standard-Statement erzeugt:

```
SELECT * from relation_name
```

Das Bearbeiten eines ODBC-Extents erfolgt dann wieder genauso, wie für ODABA2-Extents durch Konstruieren eines PI-Handles. Die Instanzen, die über das PI-Handle gelesen werden, werden entsprechend der definierten ODABA2-Structure bereitgestellt. Dabei erforderliche Konvertierungen werden automatisch durchgeführt.

7.2 Datenaustausch mit ER- und Objektmodellen

Ein komplexer Datenaustausch mit kompletten ER-Modellen oder anderen Objektmodellen ist über die DataExchange Facilities von ODABA2 möglich. ODABA2 ermöglicht die Kommunikation auf der Datenbankebene sowohl mit anderen OO-Datenbanken als auch mit relationalen Systemen, die auf dem ER-Modell basieren. Neben der Verarbeitung einzelner Relationen wird somit auch der Datenaustausch komplexer Strukturen ermöglicht.

Zu diesem Zweck kann ein Datenaustausch (DataExchange) definiert werden. Im Datenaustausch mit einem anderen System können sowohl einzelne Werte als auch Beziehungen zwischen den Instanzen ausgetauscht werden. Import- und Exportbedingungen ermöglichen die Auswahl der zu importierenden oder exportierenden Instanzen. Außerdem ist über einen Import-Handler die Bearbeitung der Instanzen nach dem Import möglich, d.h. nach dem Import einer Instanz können weitere abgeleitete Werte berechnet werden.

Der Datenaustausch basiert im wesentlichen auf der Annahme, daß alle importierten bzw. exportierten Instanzen anhand eines logischen Identschlüssels identifiziert werden können. Dadurch wird u.a. der Import/Export einzelner Felder in existierende Instanzen möglich. Ein DataExchange definiert den Austausch in beide Richtungen, also sowohl den Import von Daten nach ODABA2 als auch den Export in externe Systeme.

Um die weiteren Lösungsansätze zu verstehen - vor allem das Beispiel - wäre es sinnvoll auch das Problem als Beispiel darzustellen: welche Relationen bzw. Extents und Beziehungen existieren extern und Intern.

7.2.1 Extent-Zuordnungen

Ein DataExchange ist durch eine Reihe von Extent-Zuordnungen definiert, die den Zusammenhang zwischen internen und externen Extents definieren.

Exchange1	
ODABA2-View	**externer Extent**
Extent/elementare View	**Name**
Person	rPerson
Firma	rFirma
Person.kinder	rKinder

Dabei können auf der ODABA2-Seite sowohl einfache Extents als auch elementare Sichten definiert werden. Eine elementare Sicht ist durch eine Menge von Property-Pfaden definiert, die jeweils eine Collection adressieren.

Beispiel Der Property-Pfad **Person.kinder** definiert die Menge aller Kinder der Personen. Jede Instanz dieser Menge ist durch eine Personen-Instanz und eine Kinder-Instanz definiert. In dem Beispiel ist die referenzierte Kinder-Instanz wieder eine Person, die ihrerseits selbst auf eine Menge von Kindern verweisen kann. Durch den Property-Pfad **Person.kinder.kinder** würden also alle Enkel, deren Eltern und Großeltern bereitgestellt werden.

Personen, die keine Kinder haben, bilden keine Instanzen in der Menge des Property-Pfades. Ebenso würden im zweiten Beispiel keine Instanzen für Personen gebildet, die keine Enkel haben.

Property-Pfade beginnen immer mit dem Namen eines ODABA2-Extents. Dieser kann durch die Angabe einer Collection-Property erweitert werden, die eine Collection-Property der Structure sein muß, die dem Extent zugrunde liegt. Nun kann wieder eine Property der Structure der referenzierten Collection angefügt werden usw.

In einer elementare Sicht können ein oder mehrere Property-Pfade angegeben werden. Diese werden durch Kommata voneinander getrennt. Wenn mehrere Property-Pfade in einer Sicht definiert sind, ist die Menge der Sicht durch die Produktmenge der Property-Pfade definiert. Eine Instanz der Sicht setzt sich aus den Instanzen der definierten Property-Pfade zusammen.

Beispiel Die Sicht **Person,Firma** definiert die Produktmenge aller Personen und Firmen, d.h. aus einer Kombination jeder Person mit allen Firmen. Jede Instanz der Sicht besteht aus einer Personen-Instanz und einer Firmen-Instanz. Der wesentliche Unterschied zum ähnlichen Property-Pfad **Person.firma** besteht darin, daß der Property-Pfad nur die Kombinationen von Personen und Firmen enthält, in denen die Person beschäftigt ist (vorausgesetzt, daß die Property **firma** in Person auf die Firma verweist, in der die Person arbeitet).

Der IdentKey eines Property-Pfades ergibt sich aus den Schlüsseln der Primärindizes der einzelnen Collections. Primärindex einer Collection ist der erste definierte persistente Index für die Collection. Der IdentKey einer elementaren Sicht setzt sich aus den IdentKeys der Property-Pfade der Sicht zusammen.

Die Properties einer Sichtinstanz können durch ihre Property-Namen angesprochen werden. Sobald der Property-Name mehrfach in der Sicht auftritt, kann er durch Angabe des Sichtelementes qualifiziert werden.

Beispiel Die Sicht **Person.kinder,Firma** definiert zwei Personen-Instanzen und eine Firmen-Instanz. Sowohl die Personen-Structure als auch die Firmen-Structure hat ein Attribute **name**. Um auf einen bestimmten Namen in der Sicht Bezug zu nehmen, muß also **Person.name**, **kinder.name** oder **Firma.name** angegeben werden. Wenn als Bezug nur **name** angegeben ist, wird das erste Attribute der Sicht mit diesem Namen bereitgestellt (in diesem Fall **Person.name**).

Wenn die Sicht gleichnamige Elemente enthält (wie z.B. **Person.kinder.kinder**), ist selbst bei Qualifizierung des Namens durch das Sichtelement kein eindeutiger Bezug mehr möglich. Um dennoch den Bezug auf die Properties der Sicht zu gewährleisten, können einzelne Sichtelemente mit einem eigenen Namen versehen werden, der dann als Bezug auf das Sichtelement verwendet werden kann. Der Name eines Sichtelementes wird in Klammern hinter dem Element angegeben (z.B. **Person.kinder.kinder(enkel)** - **enkel.name** liefert dann den Namen des Kindes des Kindes).

Weitere Möglichkeiten zur Definition elementarer Sichten ergeben sich, wenn diese direkt als elementare ODABA2-Views definiert werden. In diesem Fall wird auf die Sicht über einen View-Extent Bezug genommen, d.h. im Rahmen des DataExchange wird die Sicht wie ein ganz normaler Extent angegeben und verarbeitet. Für den Import sind nur elementare ODABA2-Views zugelassen. Ist jedoch nur der Export in einen externen Extent vorgesehen, kann jede beliebige ODABA2-View als Export-Extent angegeben werden.

Jede Sicht definiert eine (virtuelle) Tabelle. Der Austausch von Daten wird nun auf der Basis der jeweiligen Tabellenzeilen durchgeführt. Dabei erfolgt die Zuweisung der Instanzen beim Import/Export anhand der im jeweiligen Zielsystem definierten IdentKeys. Über Optionen kann außerdem gesteuert werden, ob nicht vorhandene Instanzen der Ausgangstabelle in der Zieltabelle gelöscht werden sollen und/oder ob noch nicht existierende Instanzen in der Zieltabelle angelegt werden sollen.

7.2.2 Property-Zuordnungen

Beim Datenaustausch werden nur die Properties übernommen, für die explizit eine Zuordnung definiert wurde. Da der Informationsgehalt zwischen Quelle und Ziel i.allg. differieren wird, bleiben auf diese Weise bestehende Informationen erhalten.

Property-Zuordnungen werden jeweils innerhalb einer Extent-Zuordnung getroffen. Jede Zuordnung wird auf der Basis von Property-Pfaden definiert, die im einfachsten Fall aus einem Attribute-Namen bestehen.

Exchange1	
Person	**rPerson**
pid	pid
name	name
vorname[0]	first_name
vorname[1]	first_name1
vorname[2]	first_name2
adresse.strasse	street
adresse.hausnummer	number
adresse.ort.land	country
adresse.ort.PLZ	ZIP
adresse.ort.name	place
firma.fid	cid
Firma	**rCompany**
fid	cid
name	name
Person.kinder	**rChildren**
Person.pid	parent_pid
kinder.pid	child_pid

Die Tabelle zeigt die Zuordnungen, die erforderlich sind, um Firmen und Personen einschließlich ihrer Beziehung (1:N) und der Kinder-Beziehung (M:N) zu importieren bzw. zu exportieren. Im folgenden sollen die verschiedenen Fälle, die beim Import/Export auftreten können, genauer kommentiert werden.

7.2.2.1 Zuordnung der Attribute

Der einfachste Fall ist die Zuordnung von Attributes, die in den zugeordneten Instanzen definiert sind. In diesem Fall werden die Namen der Attributes, die inhaltlich einander entsprechen, zugeordnet. Sollten die Attributes auf unterschiedlichen Strukturebenen definiert sein, sind entsprechende Property-Pfade zu definieren.

Beispiel In dem DataExchange-Beispiel **Exchange1** besitzt die ODABA2-Structure Person nur ein Attribute **adresse** mit der Structure Adresse, welche die

Attributes **strasse** und **hausnummer** enthält. In der Relation **rPerson** hingegen sind die Attributes **street** (für Straße) und **number** direkt definiert. Aus diesem Grunde muß bei der Zuweisung auf der ODABA2-Seite ein Pfad definiert werden:

```
adresse.strasse      street
adresse.hausnummer   number
```

Die Personen-ID hingegen, die in beiden Structures direkt als Attribute definiert ist, kann auch direkt zugeordnet werden:

```
pid                  pid
```

Auch diese Zuweisung muß explizit angegeben werden, obwohl hier die Attribute-Namen übereinstimmen.

Eine Extent-Zuordnung muß mindestens Zuordnungen für die Komponenten des definierten IdentKeys besitzen. Ist eine Zuordnung nur für den Import bzw. nur für den Export definiert, müssen wenigstens für die Structure des Ziel-Extents alle IdentKey-Attributes zugeordnet worden sein.

7.2.2.1 Zuordnung von 1:N-Beziehungen

Auf der ODABA2-Seite der Zuordnung ist es nicht unbedingt erforderlich, daß die zu übernehmenden Attribute direkt in der ODABA2-Instanz gespeichert sind. Ebenso können Attributes in Instanzen singulärer Referenzen übernommen werden. Singuläre References oder Relationships der OO-Modells bilden dabei i.allg. 1:N-Beziehungen des ER-Modells ab.

Da die 1:N-Beziehungen im OO-Modell vielfältiger sind, als im ER-Modell, wird generell davon ausgegangen, daß eine Menge von Attributes einer Relation in eine referenzierte Instanz importiert bzw. aus dieser exportiert werden sollen. Dabei spielt es keine Rolle, ob durch diese Attributes im referenzierten ER-Modell eine 1:N-Beziehung dargestellt wird.

Beispiel Wenn der **name** in Person statt als direkt eingebettete Structure als singuläre Reference definiert wäre, würde die Zuordnung genauso, wie im obigen Beispiel erfolgen:

```
adresse.strasse      street
adresse.hausnummer   number
```

Allerdings würden auch „leere Adressen“ übernommen werden, solange dies nicht durch entsprechende Export/Import-Filter verhindert wird.

Etwas komplizierter ist die Übernahme von Attributes in singuläre Extent-basierte Relationships. Da bei der Zuordnung dieser Instanzen auf bereits existierende Instanzen zurückgegriffen wird, muß der IdentKey der Instanz zum Zeitpunkt der Zuordnung bekannt sein. Wenn der IdentKey nur aus einem Attribute besteht, ist es ausreichend, dieses Attribute als erstes zuzuordnen. Besteht der IdentKey der referenzierten Instanz jedoch aus mehreren Attributes, muß eine Unterzuordnung gebildet werden, in der die Zuordnung zu allen Attributes erfolgen kann. Die Bildung von Unterzuordnungen kann immer dann vorgenommen werden, wenn mehrere Attributes zu einer nachgeordneten Structure-Instanz zugeordnet werden sollen.

Beispiel Wenn die Structure Adresse um eine Relationship zum Ort (**ort**), in der die Person lebt, erweitert wird, und die Relation rPerson um die Angaben Ort (**ort**) und Postleitzahl (**PLZ**), kann die Beziehung durch eine einfache Zuordnung

```
adresse.ort.name      place
adresse.ort.PLZ       ZIP
```

erfolgen, solange der Ort einzige IdentKey-Komponente ist. Auf diese Weise wird z.B. die 1:N-Beziehung zwischen Person und Firma importiert bzw. exportiert, deren IdentKey nur aus dem Firmen-ID (fid) besteht.

Wird der Ort jedoch nur durch die Angaben (PLZ,ort) eindeutig identifiziert, muß eine Unterzuordnung getroffen werden:

```
adresse.ort           rPerson
  [Ort (ik_Ort)]      [rPerson]
   ort                ort
   PLZ                PLZ
```

Die Unterzuordnung ist wieder durch die Angabe zweier Structures und ihrer Property-Zuordnungen definiert.

Auch für Unterzuordnungen müssen alle Komponenten des IdentKeys zugeordnet werden, hier allerdings nur auf der ODABA2-Seite, da auf der Seite der externen Extents keine Extent-basierten Relationships unterstützt werden.

Im OO-Modell ist für Relationships i.allg. eine inverse Referenz definiert. Diese braucht jedoch nicht explizit ausgetauscht zu werden, da sie in ODABA2 automatisch gepflegt wird.

Beispiel Wenn die 1:N-Beziehung zwischen Person und Firma importiert werden soll, wird nur die singuläre **firma**-Reference in der Person importiert bzw. exportiert. Die inverse Reference **mitarbeiter** in der Firma wird automatisch gepflegt. Ein Export der **mitarbeiter**-Collection ist ebenfalls nicht erforderlich, da diese im ER-Modell nicht als eigenständige Menge dargestellt werden.

7.2.2.3 Zuordnung von M:N-Beziehungen

Einfache M:N-Beziehungen werden im OO-Modell über Relationships dargestellt. Dieses Verfahren unterscheidet sich grundlegend von den Umsteiger-Relationen, die zur Darstellung dieser Beziehung im ER-Modell gebildet werden. Durch die Verwendung von Property-Pfaden kann jedoch eine gleichwertige Sicht zwischen den Modellen erzeugt werden. Deshalb erfolgt der Import/Export von M:N-Beziehungen i.allg. über Property-Pfade in elementaren Sichten.

Beispiel Um die M:N-Beziehung zwischen Person und Kindern (Person) zu importieren, muß die Umsteiger-Relation Children des ER-Modells importiert werden. Diese wird der Sicht **Person.kinder** oder **Person.eltern** zugeordnet, die eine zur Children-Relation äquivalente Menge darstellt. In der Zuordungstabelle am Beginn dieses Abschnitts wurden zwischen **Person.kinder** und **Children** folgende Property-Zuordnungen getroffen:

```
Person.pid          parent_pid
kinder.pid          child_pid
```

Ebenso wäre ein Austausch über die Eltern-Beziehung möglich, indem der Austausch zwischen der View **Person.eltern** und **Children** definiert wird:

```
Person.pid          parent_pid
eltern.pid          child_pid
```

Es muß jedoch nur eine dieser beiden Zuordnungen getroffen werden, da sie in ihrem Informationsgehalt völlig äquivalent sind. Auf der ODABA2-Seite wird die nicht importierte Reference der Relationship (**eltern** bzw. **kinder** im zweiten Fall) automatisch gepflegt.

Sind in der Beziehung selbst weitere Daten gespeichert (z.B. die Höhe des Taschengeldes, welches das Kind vom jeweiligen Elternteil bekommt), kann der Zusammenhang im OO-Modell nicht mehr als einfache Relationship dargestellt werden. In diesem Fall wird er ähnlich wie im ER-Modell als Assoziation oder funktionaler Zusammenhang (z.B. zwischen zwei Personen, die in einer Kind-

Eltern-Beziehung stehen) dargestellt. Assoziationen werden wieder als Structures definiert, dern Instanzen in einem Extent abgelegt sind, so daß die Angaben direkt in die Assoziation importiert werden können (siehe 7.2.2.1).

7.2.3 Filter und Handler

Beim Datenaustausch kann die übertragene Datenmenge durch Filter eingeschränkt werden. Filter können sowohl auf der Seite der Quelle (Lesefilter) als auch auf der Seite des Ziels (Schreibfilter) definiert werden. Auf jeder Seite des Datenaustausches können sowohl Import- als auch Exportfilter definiert werden.

Filter werden als Funktion oder Expression der Structure, welche die Quelle oder das Ziel des Datenaustausches definieren, bereitgestellt. Damit ist die Definition von Filtern nur möglich, wenn Quelle bzw. Ziel auf einen im Repository definierten Extent verweisen. Dies kann jedoch auch ein Extent für eine elementare View sein.

Importfilter werden als Schreibfilter verarbeitet, d.h. es werden nur die Instanzen (Tupel) in die Zielumgebung importiert, für die der Filter den Wert wahr (TRUE) liefert. Exportfilter wirken als Lesefilter, d.h. es werden nur die Sätze exportiert, die der Bedingung genügen (TRUE).

Neben der Definition von Filtern ist die Definition einer Funktion oder Expression als Import-Handler möglich. Dieser Handler erlaubt die Behandlung der Instanzen auf der Zielseite nach dem Import. Auch der Import-Handler muß als Funktion der Structure definiert werden, die dem Ziel des Datenaustausches zugrunde liegt.

7.3 Direkter Zugriff auf externe Extents

Es ist generell möglich, über die ODBC-Schnittstellen externe Extents zu definieren und auf diese Weise auf andere Datenbanken zuzugreifen. Da ODBC-Extents in ODABA2 genauso wie ODABA2-Extents behandelt werden, können auf diese Weise externe Daten in die ODABA2-Anwendungen einbezogen werden. Allerdings haben ODBC-Extents einige wesentliche Nachteile:

- Sie bilden immer eine relationale Sicht ab, wodurch die Zugriffsfunktionalität verglichen mit ODABA2-Extents erheblich eingeschränkt wird.
- Sie sind Tabellen-orientiert, was erhebliche Nachteile beim Instanzen-orientierten Zugriff zur Folge hat.

Aus diesem Grunde werden einige Systeme durch spezielle Extents unterstützt. So erfolgt der Zugriff auf SDF oder Binär-Dateien über entsprechende ESDF oder BINA-Extents. Ebenso gibt es für den SQL-Server von Microsoft einen speziellen SQLS-Extent, der den direkten Zugriff auf die Daten über SQL-Server-API ermöglicht.

Durch die spezielle Unterstützung externer Extents wird die Einbeziehung externer Datenquellen auf der Instanzenebene möglich, was zu einer wesentlichen Effizienzsteigerung und wesentlich besseren Funktionalität führt. Außerdem wird es auf diesem Wege zukünftig möglich sein, auch auf andere OO-Datenbanken direkt zuzugreifen.